1998

Distributed object architectures

Santosh N. Sreenivasan
Lehigh University

Sreenivasan, Santosh N.

Distributed Object Architectures

January 11, 1998

DISTRIBUTED OBJECT ARCHITECTURES

by

SANTOSH N. SREENIVASAN

A Thesis
Presented to the Graduate and Research Committee
Of Lehigh University
In Candidacy for the Degree of Master of Science

in
Computer Science

Lehigh University
December 4, 1997

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science

12-14

Date

Thesis Advisor

Chairperson of Department

Acknowledgements

For the experience and knowledge gained over the course of my study at Lehigh University, and the production of this document, I am deeply grateful to Dr. Terrance E. Boult. Throughout this period, he provided an excellent environment that fostered the pursuit of academic knowledge and technical understanding. I thank him for the opportunity he presented, as well as his patient explanations and guidance which revealed a direction to my work at times when there seemed to be none. I take this opportunity to mention my parents - not for anything specific, since that would prove too insufficient - but for the chance to give them credit, which is not done enough! I also thank the folks at Packard 514 and 15 East 4th for the lighter side of life!

Contents

List of Figures

Abstract

Produced by the merger of the object oriented methodology and distributed systems, distributed object architectures will play a pivotal role in the future of software. The purpose of this paper is to determine the requirements for constructing an architecture which would allow objects to interact in a distributed system. To uncover these issues, three different systems are studied. First, a design developed to allow objects to communicate over process and machine boundaries using the Message Passing Interface (MPI) is presented. Both OMG's CORBA and Microsoft's DCOM are then examined in order to study the methods adopted by comprehensive distributed object architectures such as these. By studying these three systems, the salient issues involved in creating a distributed object architecture are identified. Lastly, a quantitative analysis of each of these three methods is presented, with respect to various factors. Although distributed object architectures present several advantages over the traditional approach, this analysis reveals that the overhead of large architectures like CORBA and DCOM prevent these systems from utilizing higher bandwidth connections as well as light weight network programs.

Chapter 1

Introduction

In the history of computing, there have been a few major leaps in technology that redefine methods of working and approaching a problem. In the software arena, two such leaps were triggered by the advent of object oriented computing and the shift from centralized to distributed systems.

The object oriented approach to software development allows the modelling of real-world entities as objects that are accesed through clearly defined interfaces. Using the concept of encapsulation, this allows flexibility for the object to evolve over time, without affecting the clients who depend upon the services of this object. Objects can also be organized in hierarchies, allowing the expression of complex relationships. This is made possible by two more prongs of object oriented methodology - inheritance and polymorphism. The advantages in this approach have been proven and accepted in today's software world. By allowing the reuse of objects, and promoting an effective style of designing applications, the object oriented approach to development has also been embraced by the software industry.

Another emerging trend is largely triggered by economic factors. As computing power and memory become cheaper, the movement away from centralized systems with large servers is rapidly gaining momentum, leading towards distributed systems comprised of individual workstations. This is fueled by growing network capabilities, represented by improved switching methods and advances in physical transport media. The growing dependence on networks is evident in the

prevelance of the Internet in modern life and daily activities, spanning areas such as leisure, business, technology.

In order to unleash the potential of today's networks, software needs to be modified to fit the distributed paradigm. Single-machine systems are becoming increasingly limited in their utility, and will soon be abandoned for their distributed successors. To meet this demand for developing distributed software, the object oriented approach emerges as obvious candidate, because of the proven advantages it offers. The marriage of these two worlds has produced a new breed of software, known as the distributed object oriented system.

There have been several attempts to establish an architecture for developing distributed object systems, some of which have been driven by major industrial forces. The purpose of this paper is to determine what is needed to construct an architecture which would allow objects to interact in a distributed system. An introduction to the complexities of this problem are presented in the first part, which focuses on a rudimentary approach that we developed, allowing objects to communicate across process and machine boundaries. The second part of the study focuses on a comprehensive architecture that is widely used in the software industry, namely CORBA. Following this, the third part presents CORBA's major comptetitor, the COM/DCOM model, which adopts a different style, to present an approach to distributed object oriented development.

We also require a way to assess these systems. Andrew Tannenbaum has identified the following design issues for distributed systems [Tan95]:

- Transparency
- Flexibility
- Reliablity
- Scalability
- Performance

To this, we add our additional criteria for assessing the three approaches covered in this study:

- Ease of development

- Compatibility

A practical implementation of each system will also be presented in each part, leading to various timing tests which will allow us to compare these systems in a quantitative mesure. Finally, we will have addressed the requirements of building a distributed object architecture, assessed each of the three methods covered in this study in lieu of the criteria above, and presented a quantitative comparison of their performance.

Part I

MPI

Chapter 2

Overview of the Message Passing Interface (MPI) [1]

The task of distributed object oriented computing can be considered at various levels, from application interaction down to the networking aspects, not to mention the various layers that are needed in between. At the most fundamental level, the challenge is to synchronize different processes to work together in a seamless fashion. Objects that do not share a common address space must effectively and accurately communicate across process boundaries. In addition, independent processes must be given a "feel" for other processes that it must cooperate with.

If we can tackle this fundamental requisite of enabling processes to work together in synchrony, then the weightier task of designing a distributed system can be examined without impediments of lower level detail. This abstraction of the "communication" layer is why the Message Passing Interface was chosen for this study.

The Message Passing Interface (MPI) [2] is a specification for a standardized and portable message-passing system initially designed for use in parallel computing environments. The basic function of this standard is to define a widely implementable method for passing messages between pro-

[1] This chapter was based largely on material from [SOHL+96]

[2] The full specification can be acquired from http://www.mcs.anl.gov/mpi/index.html.

cesses, to enable them to communicate in a reliable manner. With this means of communication, processes that are separated on a single machine, or over different machines, can cooperate in the solution of a problem.

We will first outline the salient features of this standard, and then examine the details of having objects communicate between processes. We shall also introduce a few implementations of this standard, including two that were used in this study to develop our mini-system. Lastly, we shall apply our knowledge of MPI to our goal of understanding the requirements and design of a distributed object-oriented system. To further elucidate this, the next chapter contains an implementation overview of using MPI to enable objects to communicate over process boundaries.

2.1 History of MPI

Systems that enable message passing between processes have been in use for a long time. Considerably before MPI was conceived, several vendors have implemented their own variant of this paradigm. The MPI Forum, consisting of over 80 people from 40 organizations, was formed to standardize this effort and define a formal syntax and semantics of a standard core of library routines that implement a message passing interface. This standard has the advantage of being able to draw from the best features of many existing message passing systems. Implementations of MPI are available on a wide range of hardware platforms from individual workstations to parallel computers, and a wide range of operating systems.

MPI can be used by multiple processes on the same machine, or by processes that are separated across machine boundaries. It is also possible to span heterogeneous systems, that include machines with different architectures and communications protocols. MPI takes care of these differences in a relatively transparent manner, besides the fact that the MPI implementation must support this heterogeneity. This is a big advantage in todays networking environments which invariably contain different types of machines and operating systems. The MPI standard supports both Multiple Program, Multiple Data (MPMD) programs, in which each process follows a unique execution path as well as Single Program, Multiple Data (SPMD) programs, where all processes follow the same execution path.

The original standard was released as version 1.0 in June 1994, and was modified to version 1.1 in June 1995. We shall be covering version 1.1 in this study.

2.2 Basics of MPI

The MPI standard basically defines an application programming interface, that comprises different types of inter-process communication, and modes of operation. The functions in this interface are specified in a language independent way. MPI arranges processes in groups, called communication domains. Processes in a group are ordered, and identified by their rank, which is represented by an integer number. Communication can be intra-group or inter-group, as we shall see. Also, processes may participate in several domains. This means that domain boundaries may contain overlapping processes.

Messages carry information that is used to distinguish and selectively receive them. Called the **message envelope**, this consists of:

- **source:** identity of the sender
- **destination:** the range of valid values is 0,...,n-1 where n is the number of processes in the group (Its rank).
- **tag:** this can be used by the application to distinguish types of messages, and selectively receive them.
- **communicator:** Representing a communication domain, this is a global structure that allows communication between processes within a group (intra-communicator), or processes in different groups (inter-communicator).

MPI calls also have the following characteristics:

- **local:** the completion of the call only depends on the local process, and does not need explicit communication with another user process.
- **non-local:** the completion of the call may require the execution of some MPI procedure on another process.

MPI Data type	C Data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Figure 2.1: C mappings for MPI Data types

- **blocking:** a return from the call indicates that the user is free to re-use the resources specified in the call.
- **non-blocking:** the call may return before the operation completes, and before the user is allowed to re-use the resources of the call.p
- **collective:** all processes in a process group need to invoke to procedure.

2.2.1 Point-to-Point communications

The basic communication mode of MPI is transmittal of data between a pair of processes. One process sends a message and the other one receives it. Although seeming very simple, there are many variations to this theme that provide a comprehensive suite of communication methods. While it is not the intention to detail the usage of these methods, the following section will provide some insight into the capabilities of MPI communication.

Simple Blocking Calls

A blocking send is performed by the MPI_SEND() function, that sends a fixed number of successive entries, all of the same data type. The data type is indicated by basic MPI types, which map to data types of the host language, as shown in figure ??.

counterpart to this is MPI_RECV(), that will receive a fixed number of succesive entries of the same data type. This can be initiated at any time, and will block until it receives the data. In addition, the receive call can receive from any arbitrary sender, while a sender must specify the destination process. This basically allows for the transfer of a pre-determined number of elements which must be of the same data type, and physically located in contiguous memory locations. More flexible ways of point-to-point communication are covered later.

The transfer of a message comprises three steps:

1. Data is copied out of the send buffer and a message is assembled
2. A message is transferred from the sender to the receiver
3. Data is copied from the incoming message and disassembled into the receive buffer

There is strong emphasis on type matching at each of these steps, to ensure operability over process boundaries. The data type of each element in the sending buffer must match the type of the entry in the send operation. Also, the type of the send operation must match the type of the receive operation, and lastly, the type of the receive operation must match the type of each element in the receive buffer. To handle heterogeneous architectures, MPI requires that the representation of data types be changed appropriately when messages cross machine boundaries. This accounts for differences in word length and byte ordering between different types of machines.

In addition, messages between MPI processes follow the order in which they are sent by the sender, and received by the receiver. In other words, they are non-overtaking. Figure 2.2 illustrates this principle.

Simple Nonblocking Calls

A common method for improving performance is to overlap communication and computation. This is achieved by using nonblocking calls. Instead of waiting for the call to complete, the caller is free to perform other tasks. A nonblocking post-send initiates a send operation, but does not complete it. A separate complete-send call is needed to complete the communication, and verify that the data has been copied out of the send buffer, qualifying it for reuse. The counterpart

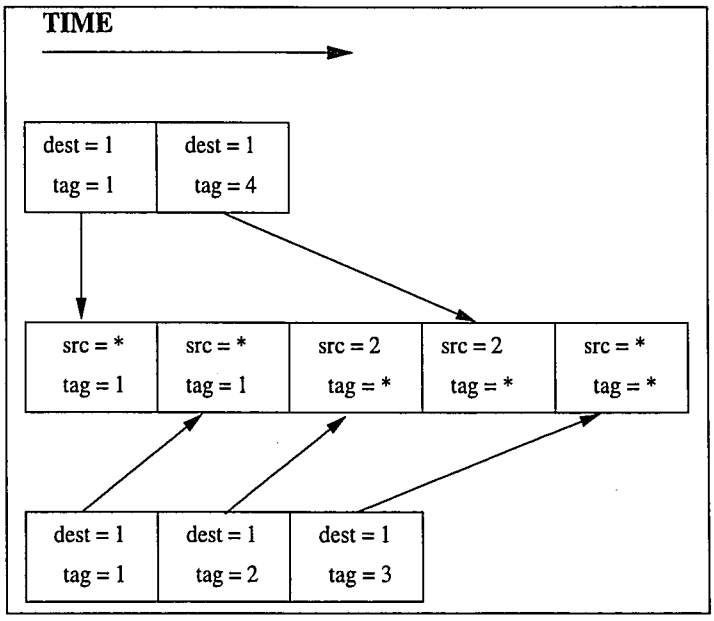

Figure 2.2: Messages received in the order they were sent [SOHL+96]

of these are post-receive and complete-receive calls. These nonblocking calls can interact with blocking calls on the other end. For instance, a nonblocking send can be used with a blocking receive in the other process.

Posting operations are implemented by the MPII_SEND() and MPI_IRECV() functions, and the operations MPI_WAIT() and MPI_TEST() may be used to complete both nonblocking send and nonblocking receive calls. Nonblocking calls are also non-overtaking.

It is possible to complete multiple pending nonblocking calls with one call. MPI_WAITANY() and MPI_TESTANY() will complete any one of several pending operations. MPI_WAITALL() and MPI_TESTALL() will complete all of a list of pending operations. In addition, there are calls to poll incoming messages without actually receiving them, which gives the application the flexibility of deciding what to do with a message. For instance, one could allocate appropriate memory according to the length of an incoming message, providing efficient run-time memory usage.

Persistent Communication

In many situations, repeatedly opening a communications channel to send data proves to be very inefficient. MPI allows the creation of persistent requests, that bind once to the list of arguments that need to be sent, and can be repeatedly used to initiate and complete messages. The amount of data transferred between sender and receiver is the same, but the overhead of creating and removing connections by using send and receive calls is eliminated. MPI_SEND_INIT() and MPI_RECV_INIT() create the persistent communication requests. The actual communication is initiated by MPI_START(), or MPI_START_ALL() for a list of persistent requests. MPI_CANCEL() may be used to cancel both persistent and ordinary communication requests.

Communication Modes

So far, the methods we have discussed all used the **standard mode**, in which it is up to MPI to decide whether to buffer outgoing messages or not. In this mode, the send call can be started whether or not a matching receive has been posted. If it does buffer them, then the send call might complete before a matching receive is called. MPI might not buffer the outgoing message for performance reasons, or because of lack of space in memory. In this case, the send will not complete until the matching receive is called, and the data has been moved to the receiver, freeing the buffer of the send call.

In **buffered mode**, a send operation can be started whether a matching receive has been posted or not, as in standard mode. Also, it too might complete before a matching receive is posted. To complete the operation, it might be needed to buffer the outgoing message locally. In this mode, the buffer space is provided by the application, and an error will occur if there is insufficient buffer space for the message. This space is freed when the message is transferred to its destination, or when the send call is canceled.

A send call in **synchronous mode** can also be started whether a matching receive has been posted or not. However, the send will only complete when the receive has been posted and it has started to receive the message sent by the synchronous send. Communication does not end at either side before both processes rendezvous at the communication.

Ready mode stipulates that the send call may only be started if the matching receive has already been posted. If not, the operation is erroneous. This can potentially be used to remove a handshake between two processes, hence improving performance.

In each of these modes, MPI supports both blocking and nonblocking calls, as well as persistent requests.

2.2.2 User-Defined Data types

Now that the basic operation of MPI has been outlined, it is time to consider more practical applications of this standard. Although all program logic is built upon the atomic or primitive data types such as integers, floats and characters, almost all applications combine these types into more complex aggregations that closely mirror items in the real world. It is essential to have this capability to produce understandable code, store data in a relevant manner and perform complicated tasks involving these structures.

Messages in MPI can be composed of either basic data types or user-defined types that are aggregations of these basic types. As we shall see, there is considerable flexibility in defining the layout of these data types to suit one's needs.

A user-defined or **derived** data type specifies two things:

1. A sequence of primitive data types.

2. A sequence of integer(byte) displacements.

The displacements need not be positive, distinct, or in increasing order, allowing them to be scattered in actual memory. Pairs like these are referred to as **type maps.** A handle to a derived data type can be used in place of a primitive type in send and receive operations.

There are several functions in the MPI standard that can be used for constructing derived data types.

Contiguous

The MPI_TYPE_CONTIGUOUS method is the most basic data type constructor that replicates a data type into a specified number of contiguous locations. Figure 2.3(a) graphically portrays this. It returns a handle to the newly formed type which can be used in communication operations.

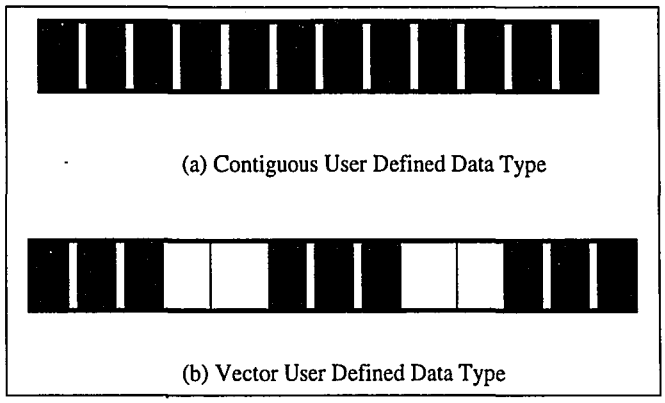

Figure 2.3: Contiguous and Vector constructed types [SOHL+96]

Vector

MPI_TYPE_VECTOR allows replication of a data type into locations that consist of equally spaced blocks. The number of blocks, elements in each block and the space between each block, or the stride, can all be specified by the user. This is shown in figure 2.3(b). The stride is measured in the number of elements between blocks.

Hvector

MPI_TYPE_HVECTOR is the same as the previous call, except that the stride between successive blocks is measured in bytes

Indexed and Hindexed

This allows the construction of data types in which the data is noncontiguous and the displacements between successive blocks need not be equal, as shown in figure 2.4. This offers

more flexibility to the user, by being able to scatter the elements, for instance, to specify different components of an array. Hindexed again measures displacements in terms of bytes, instead of the number of elements. The function calls for these are MPI_TYPE_INDEXED and MPI_TYPE_HINDEXED.

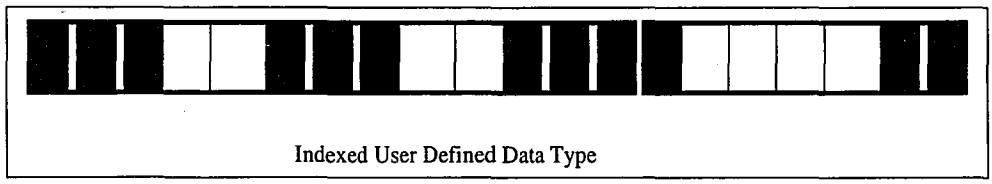

Figure 2.4: Indexed derived data type [SOHL+96]

Struct

This is the beginning of truly flexible data types that allow combinations of different primitive types into a single user-defined type. MPI_TYPE_STRUCT is similar to MPI_TYPE_HINDEXED, except that each block can contain replications of different types. The displacement between blocks is measured in bytes.

2.2.3 Use of User-Defined Data types

Once a data type is constructed using the above methods, it must be committed before it can be used in a communication method. MPI_TYPE_COMMIT does this operation, after which the data type can repeatedly be used to communicate different data. The data type is deallocated by calling MPI_TYPE_FREE.

Type matching is accomplished by examining the type signature of the participating data type objects. This is basically the sequence of primitive data types without regard to displacements in between. There are other methods to support the communication of derived data types, to find the length of a data type, or the number of elements in a message.

2.2.4 Pack and Unpack

Messages can also be explicitly packed into a contiguous buffer before sending by the MPI_PACK method, and unpacked from a contiguous buffer that is received using the MPI_UNPACK method. The difference between the derived data type approach and this approach and is that the former specifies the layout of the data to be communicated, after which MPI directly access a noncontiguous buffer to form the message. Conversely, these methods explicitly arrange the message in a contiguous buffer.

The pack and unpack functions provide for compatibility with existing message passing libraries that use this method, and also allow a message to be received in several parts, where the receive operation done on a later part might depend on the content of a former part.

Derived data types are good to use in cases where the data layout is defined by program declarations, such as structures, or where the layout is regular. In these cases, this method will avoid data copying, and the information of the layout can be reused for different data. For complex or dynamic types, however, describing the layout could be quite difficult, in which case it would be easier to explicitly pack the data into contiguous buffers. Packing could also be more efficient when the receiver acts differently according to the layout of the transmitted data.

2.3 Groups of Processes

The true purpose of such an intensive effort to define a standard for message passing was to enable cooperation and the sharing of information between multiple processes in a group, not just to have a pipe with the two ends of a sender and a receiver! MPI provides extensive support for inter-group and intra-group communication, which coordinate multiple processes.

2.3.1 Collective Communication

This type of communication relies upon an intra-communicator object, which basically specifies a communication domain, or the processes that are in a group. A group is an ordered set

of processes in which each process is associated with an integer rank (0,...,n-1 where n is the number of processes in the group).

Collective functions only have a blocking mode, and do not use a tag to identify types of messages. Therefore within each intragroup communication domain, collective calls are matched according to the order of their execution. In addition, collective functions only have one mode which is roughly similar to the standard mode of point-to-point communication.

A collective operation is performed by having all processes in the group call the communication routine with matching arguments. User-defined data types are allowed in collective communication. Following are the types of collective communication functions.

Barrier Synchronization

The method MPI_BARRIER implements this type of communication by blocking the caller until all group members have called this function. In any process, the call only returns after all processes in the group have called this function. This is illustrated in 2.5.

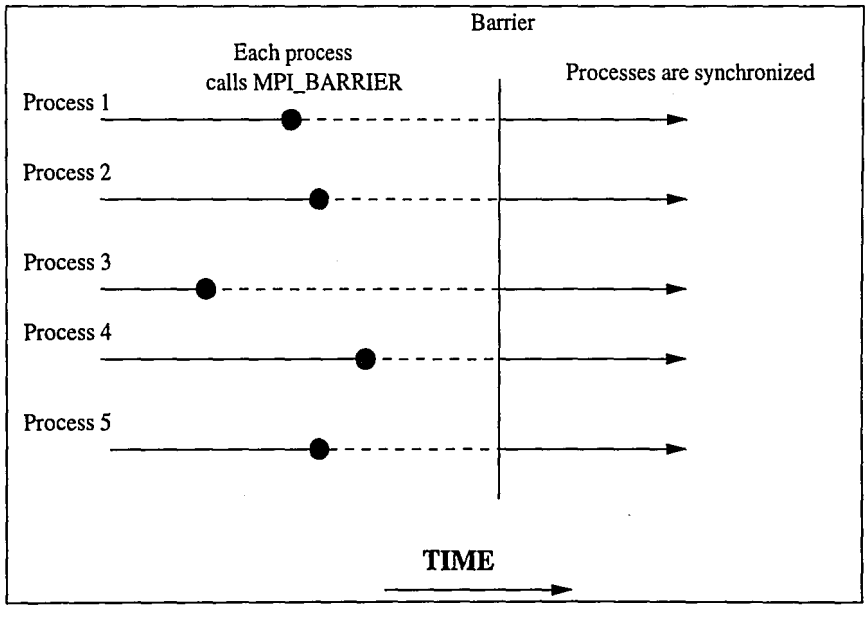

Figure 2.5: Barrier Synchronization

Global Communication

- Broadcast This simply allows a message to be sent from one process (the root process) to all processes of the group, and is called MPI_BCAST. There is only one root process in a group, and the result of this call is that the root process's communication buffer is copied to all processes in the group. This can message can be composed of any sort of data, including that of derived type.

- **Gather:** Here, each process including the root process sends the contents of its send buffer to the root process, which receives each message and stores them in order of the rank of the sending process. This is implemented by the method MPI_GATHER. It is also possible to gather a varying amount of data from each process, by using MPI_GATHERV.

- **Scatter:** This is the reverse of the above operation, in that the root splits its send buffer into n equal parts and sends the ith part to the ith process in the group of n processes. Like the gather function, it is also possible to send segments of varying sizes to different processes in the group. MPI_SCATTER and MPI_SCATTERV are the two methods that implement this. Derived types are allowed in all gather and scatter functions.

- **Gather to All Members:** This is the same as MPI_GATHER, except that all processes receive the result, not only the root. This can also be done with varying amounts of data going to different member processes.

- **All to All Scatter/Gather:** In this form, each process sends distinct data to each of the receivers. This results in a complete exchange of information between the processes of a group.

Global Reduction

This section of collective communication methods focuses on operations such as sum, max, logical AND, min, etc. that may be performed across all members of the group. It is also possible to define one's own operation, and perform it globally within the group. These functions can return the result at one node, at all nodes, and do a scan function, that returns to process i the result

of the reduction in the send buffers of processes 0,...,i (inclusive). A reduce-scatter also combines the effect of reduction and scatter functions.

Operations that are supported are:

- maximum
- minimum
- sum
- product
- logical AND, OR, or XOR
- bitwise AND, OR, or XOR
- max value and location
- min value and location

A user-defined operation can be created by using MPI_OP_CREATE.

Using Groups of processes

Although this feature was not used in my implementation, it is potentially very useful, especially in the realm of distributed objects. Imagine if a group of processes existed, each offering a particular service. With intragroup communication, a process that needs to access a service could query all others in the group, and get what it needs. It could also offer a service to all members of the group, such as synchronizing times, if it is in charge of the clock object. Extend this idea to intergroup communication, and it is conceivable to have entire systems of processes communicating with one another, and cooperating towards a common task.

2.4 Lower Layers

MPI assumes a reliable transmission of messages between processes. At the lower level, this is achieved by spawning multiple processes and establishing connections between them. The

processes can either be created on a single machine, or on different machines interconnected by a network. Depending upon the implementation, the connections between processes of a group can be made using TCP/IP sockets, shared memory or other system-specific mechanism. The implementations used in this study, which are introduced below, use TCP/IP socket connections between the processes they create.

2.5 Implementations of MPI

There are many implementations of the MPI standard that are available, and a list of these is maintained at http://www.osc.edu/mpi/. Most of these fully implement the MPI 1.1 specification, which includes all the features we have discussed above.

The implementation chosen to develop our mini-system is MPICH, supplied by Argonne National Laboratory and Mississippi State University. This supports the greatest number of platforms, and operates over heterogeneous collections of workstations. However, since MPI only supports C and FORTRAN 77, we need another way to utilize object oriented methodology for our system. This was made possible by an an extension of MPI, developed by the Department of Computer Science and Engineering at Notre Dame University. This system presents the functionality of MPI as an object oriented class library. Called the Object Oriented Message Passing Interface (OOMPI), this system allowed us to construct our miniature distributed object system.

2.6 Reason for using MPI

The reason MPI was chosen as part of this study was to allow us to examine the issues in supporting objects that are distributed across several processes. MPI takes care of this problem at the lowest level, by simply providing a channel of communication. Additional "infrastructure" must be built on top of this layer to support remote invocation of methods, object migration, and other necessary functions, as we shall see in the next section.

Chapter 3

MPI Implementation

As stated earlier, the main purpose of this part the study was to examine the needs of a distributed object oriented system. The best way to do this is to isolate the communication, which is truly unrelated to the problem, into a separate, transparent module, which is provided by the Message Passing Interface.

3.1 Description of problem

With MPI as the layer for communication between processes, we can shift our focus to the various services that will be necessary in a distributed object environment. One of the primary advantages in using the object oriented methodology lies in being able to abstract away the details of implementation from the user's or programmer's view. To facilitate this, it must be possible to call a method of an object, and receive the proper response. The caller's only view of the object is presented through the collection of methods it implements. We will use C++ as the language of implementation for our system.

To perform this invocation, the calling party, henceforth referred to as the client, must have a way to refer to the target object. When two objects reside in the same process, this is a trivial matter of a function call, since they share the same address space. However, each executing program is

given its own distinct address space. Making this invocation becomes a serious problem when they are split over different processes. Consider the figure 3.1

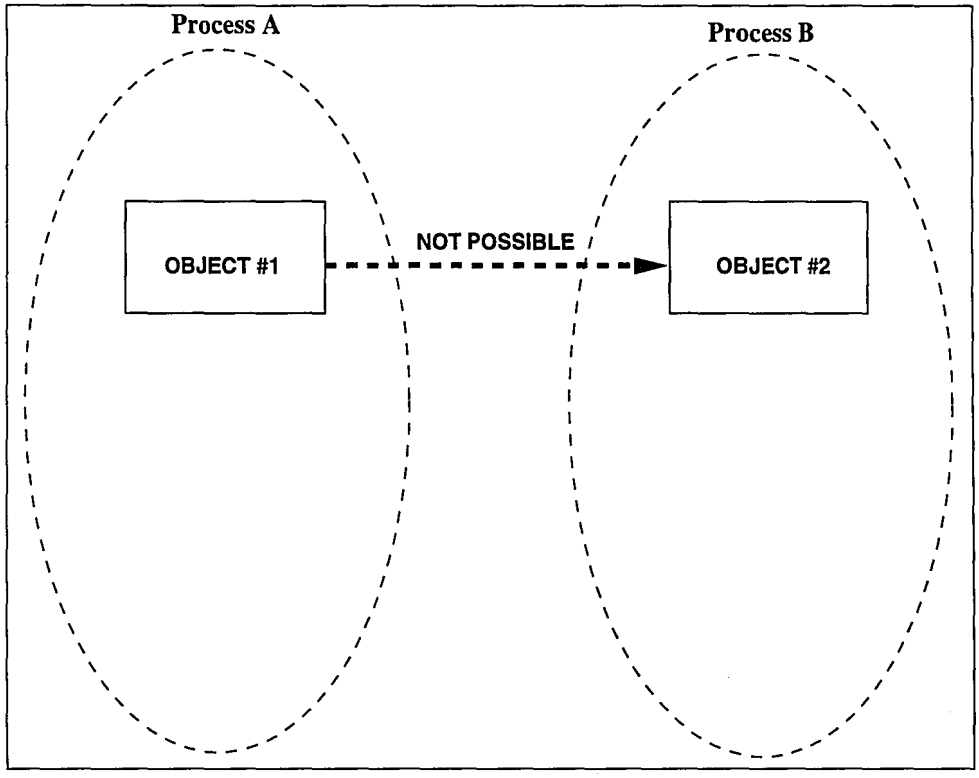

Figure 3.1: Objects in different processes cannot communicate

Instinctively, one way that this invocation can be made possible is if object 'B' could be made to exist in object 'A's process. This can be accomplished by replicating object 'B' in the same process as object 'A'. Ignoring the side-effects of multiple identity of objects for the time being, this will produce the desired result by allowing them to share the same address space. In addition, it will avoid the situation of having to carry a request and its response across the network. Object 'A' will invoke a message of the newly created object which is a replica of 'B' and will receive a response as if object 'B' had itself answered the request. This is portrayed in figure 3.2.

Although simplistic in its approach, even this proves to be a tricky task. Replicating an object in a different process requires considerable work, as we shall see. The *state* of an object can quite easily be transferred from one process to another, by filling a structure with the object's state, and then passing this structure using any communication methods. However, an object

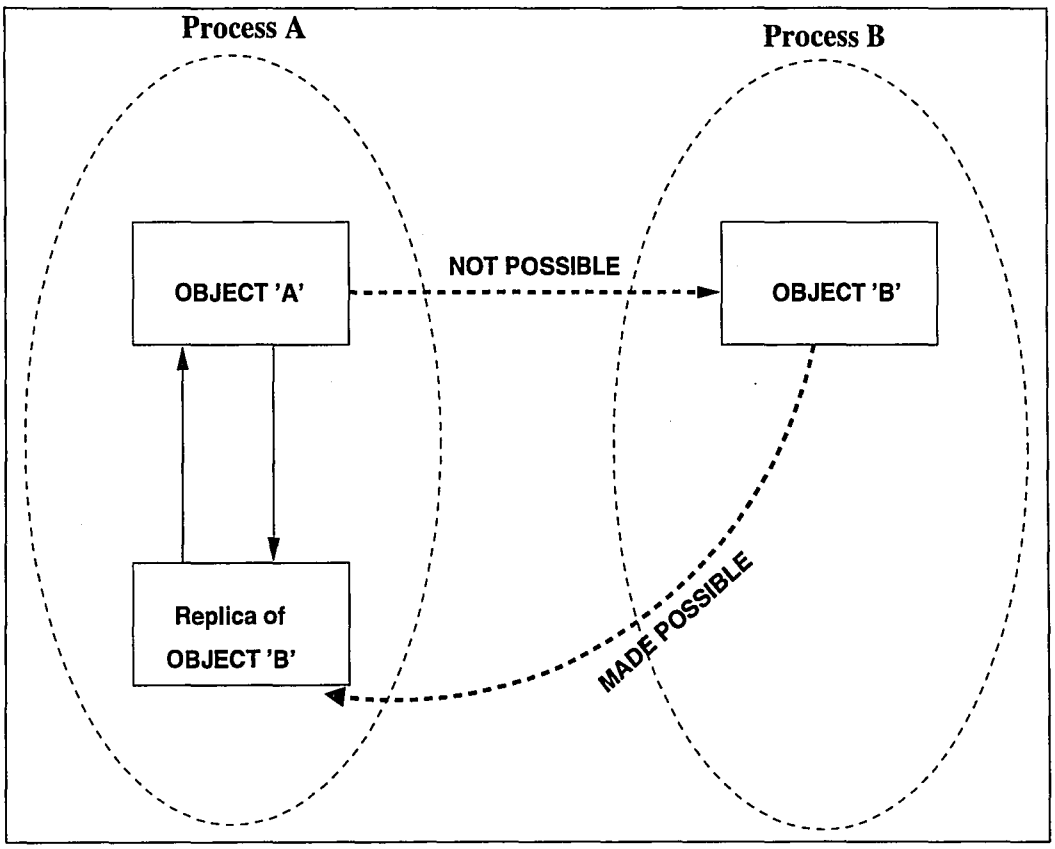

Figure 3.2: 'A' communicates with a replica of 'B'

comprises both *state* and *behavior*, proving this solution incomplete. A further complication arises when there are more complicated objects that use composition and contain sub-objects, or have references to other objects. References are implemented as pointers in C++. In a different address space, a pointer has no meaning, and could even have damaging results. Furthermore, since we are trying to establish an entire distributed object system, it is a practical requirement to be able to add new objects into the system. Our solution should be extensible.

3.2 Problem evolves

To transfer both state and behavior of an object, we can copy it to another instance of the same type. To facilitate this, we need "empty" objects that are not in use, so that their state can be filled. Creating extra dummy objects at startup in the target process for this purpose would

types of objects in the system? If these dummy objects are never used, it is a waste of time and memory.

We apparently need a way to dynamically create instances of an object in the target process. Since our system should be extensible to new objects, a further requirement is that we are able to instantiate any arbitrary class in the system. Referring to 3.2, once this new object is created, then it can be filled with the state of the object 'B', making it a true replica.

To instantiate an object, the target process must have access to its class declaration. Another simplistic solution is found in using a Single Program Multiple Data (SPMD) paradigm. In this way, every process shares the same executable image, and hence contains the same declarations. Since we will need the ability to replicate any arbitrary type of object, our program cannot presume any prior knowledge about which class to instantiate. This means that the classes in our system should be capable of instantiating themselves, and produce a new object of its type. If we think about how this member function is called, we have the typical chicken and egg conundrum! (A member function can only be called by having an object to handle the function request.) This problem is solved by requiring that each class have a **static** member function that creates and returns an instance of itself.

To review, it is now possible to dynamically create an instance of any arbitrary type of object in another process (assuming a SPMD mode of operation). With this in place, the steps of a remote invocation are as follows:

1. A client wants to make an invocation on a remote object.

2. An instance of the same type as the remote object is created in the client's process, using the class's static instantiation function.

3. The state of the new instance is filled with the state of the remote object, making it an exact replica.

4. The client makes the invocation on the local replica and gets the same response that it would get from the original object.

This is a large leap from being confined within a single process! However, there are still many details to consider to make this possible. First of all, there may be many types of objects in the system. How does the receiving process know which class to instantiate?

3.3 Registry of Objects

It is almost certain that any system would need to support more than one type of object. To perform the arbitrary instantiation that we discussed in an extensible manner, we need a management layer that would handle requests for instantiation and call the static instantiation function of the proper class. A practical example would illustrate this point.

Consider the example where there are the following three types of objects in a system:

- Person
- Animal
- Vehicle

An object in the remote process can be any one of these three types. The client, in a different process, decides that it wants to invoke the display method of a remote Honda object, of type vehicle. This means that the Honda object has to be replicated in the client's process. However, the problem is that the client process does not know what type of object will need to be replicated at runtime. Assuming that the object can be brought over, what will the target process do with it? It is unreasonable to have the client create the object it desires, and then fill its state with the remote Honda object. This would simply shift the problem to the client, and our distributed system would not be of much use, since one of our criteria is transparency.

It is also not feasible to have a hard-coded switch statement like figure 3.3 since this is not extensible. To add a new class of objects into the system, the entire program would have to be recompiled, which is far too cumbersome and inefficient.

Two things are necessary in this situation. First, we have established that each class must have a static member function that instantiates the class and creates a new object. Secondly, to handle

```
switch (type) {
case PERSON: { //code for person }
case VEHICLE: { //code for vehicle }
case ANIMAL: { //code for animal }
}
```

Figure 3.3: Switch statement

this dynamic instantiation of objects in the system, a registry object is needed which knows the address of the static member function of each class. This way, when an object is brought over, the registry object recognizes which class of objects it belongs to, and creates a new instance which will be the replica. The client transparently invokes the replica's function.

The registry needs to maintain the information about static instantiation functions of each class. This can be implemented by creating a mapping from a string description of the object to the address of the proper instantiation function. Now when the state of an object is brought over from another process, a unique string description can also be carried with it. The registry object looks up the description in its tables, finds the class which the object belongs to, and creates a new object of that type. Then this new object can be filled with the state of the remote object, making it a true replica.

By introducing a middle layer between the communication and the object interaction, we have been able to make this an extensible and feasible solution that is transparent to the client. Each object can be made to register with the registry object at startup and inform it of its unique description, as well as the address of its static member function for instantiation. Thereby, the registry knows all the types of classes in the system, and is able to dynamically create an object of any arbitrary type that is needed.

To ensure extensibility, the process of looking up a description and calling the proper member function must be generic, and not contain any class or type-specific information. That is, this process cannot return a pointer to a vehicle object, or a person object. This process must necessarily return a void pointer to the object, which is type casted by the client in the appropriate manner. This way, any type of object can be returned, even ones that are added into the system later on.

```
class vehicle {
  char license [8];
  int  wheels;
  char make [20];
  char model [10];
  int grade;
public:
  vehicle (char * lic, char * mak, char * modl, int wh=4, int gr=1);
};
```

Figure 3.4: Declaration of Vehicle class

3.4 Communicating State of Objects

Now that we know how to duplicate an object on another process by transferring its state into a newly instantiated replica, we have to attack the next problem. How is the state of an object transferred? Referring to figure 3.2, we have assumed so far that there is a way to transfer the state of object 'B' from the native process to the target process. This seems to be simple in the case of the generic examples we discussed. The data members of the vehicle class in figure 3.4 are relatively simple, and can be easily copied into a structure with similar members. This structure can then be passed to process 1 using the pack function of MPI, or its derived data type capability.

However, this example is naturally too simplistic to be of use in any real-world application. It is common for classes to have pointers or references to objects as data members. In the receiving process, neither the pointer nor the reference would have a use-able value. In addition, two fundamental features of object oriented design will cause serious problems with our way of copying data members into a struct. Namely, these are inheritance and composition.

If class 'A' is derived from class 'B', objects of type 'A' contain all of the characteristics of the base class 'B', except in cases of private inheritance. In this case, how will we ensure that an object is fully described and all of its contents are properly transmitted to the receiving process, especially in the case of large inheritance hierarchies? Furthermore, classes are often composed of other classes. This leads to a recursive description of classes as members of containing classes, and could be quite complicated.

3.5 Data Exchange Standard

It is obvious that we need a reliable way of describing the state of objects when they are transferred from one process to another. This should be capable of handling complex objects that refer to other objects, and also utilize the features of an object oriented design such as inheritance and composition.

The Data Exchange Standard (DEX) is a tool that has been designed for such a task by the architects of the Image Understanding Environment (IUE). This system was originally established for the purpose of exchanging object descriptions between Image Understanding systems. It has however been generalized for use in any system that needs to describe the state of complicated objects. The basic function of DEX is to provide a character-based format for describing objects. This format, arranged in a Lisp-like syntax, provides a portable and understandable way to represent an object that is otherwise stored in binary format. A system can use this standard to describe its objects, and then exchange this description with any other system that understands the DEX format.

Using DEX, it is possible to read the state of an object, and write it out as a character stream, which is easily processed by any system. Since the output is made up of simple characters, and a Lisp-like syntax is used, it is even readable by users, and can be modified by simple editors. The medium for input and output in DEX is a stream. An object description can be written out to a stream, and can be read in from a stream.

There are two separate APIs provided with DEX [spe]. The Generic API supports

- production of the standard representation given a software table based description of the objects
- reconstruction of the table description from the standard representation

The Object Oriented API supports

- the instantiation of C++ objects in a program directly from DEX
- the construction of DEX descriptions from a set of C++ objects within a program

By using the DEX format, it is possible to express the state of any object in our system. This description can then be passed to the receiving process using MPI. This is easily accomplished since the description is nothing but a series of characters. From MPI's point of view, it is simply passing an array of characters between processes. An example using the Generic DEX API is discussed next, followed by the implementation of the Object Oriented DEX API to communicate complex objects in the IUE.

3.6 Generic DEX example

We will extend the definition of the example classes presented earlier. To fit into our system, we will require that these classes be capable of

1. Writing their state to a DEX stream
2. Filling their state from a DEX stream
3. Instantiating themselves to produce a usable object

The third function is handled by our registry object. The other two tasks could be performed by the DEX Object manager, as is seen in the next section. However, with the generic DEX API, it is necessary that this functionality be transferred to the users objects. A compromise is seen here between requiring too much complexity from user objects, and the task of writing an object manager for the system. For few objects, it is reasonable to adopt the method outlined here. As the number of objects in the system grows, it would be wise to develop an object manager to perform these functions for the objects and allow the objects to be simpler.

Each of these tasks should be handled by a function with a particular name that is known to the registry object. This allows the addition of any number of new classes into the system, as long as they follow these rules. Figure 3.5 shows the names of these functions, namely fill_dextable_from_obj() and fill_obj_from_dextable(). Again, it is not important what these names are, but just that they are standardized across all objects in the system and the registry object knows these names.

```
class vehicle {
  char license [8];
  int wheels;
  char make [20];
  char model [10];
  int grade;
public:
  vehicle (char * lic, char * mak, char * modl, int wh=4, int gr=1);

  static void * create_instance();
  static void fill_dextable_from_obj (vehicle & vehicle_obj,
      DEX_table & table,
      DEX_class & cls,
      DEX_object * obj,
      int tb_count);

  void fill_obj_from_dextable (DEX_table & table, int pos);
};
```

Figure 3.5: Full Declaration of Vehicle class

A simple example of a system that uses this approach is seen in Appendix A.1. The objects in the system write themselves out to a character stream in the DEX format. A string is constructed from this stream and transferred to the client's process using the Object Oriented MPI. In the client's process, the string is received, and bound to an input stream for DEX. This string contains the full description of the object, including a short name indicating the type of the object.

This name is inspected, and the registry object returns the static instantiation function of the proper class. An object of this class is created, and it fills its state from the DEX stream. A true replica of the object now exists in the same address space as the client. The client proceeds to invoke one of its member functions.

3.7 Object Oriented Implementation

For larger systems with many more objects than we presented in our example, the Object Oriented API provides an efficient way to describe the state of any object in the system. The Object Oriented API is a layer above the Generic API, and uses its functionality to hide lower level

details and provide a way of converting between objects in a system to information in the DEX files. To tailor this to a particular system, it uses an Object Manager that handles instantiation and description of the systems objects appropriately.

The Object Manager has to be specifically written to handle objects in a system. The DEX standard specifies what information it needs to incorporate to perform the functions of writing an object's state into a DEX description, and instantiating objects using these descriptions. Using this standard, and by customizing the Object Manager to the needs of our system, it is possible to use DEX to represent the state of any object in the system. The advantage in using the Object Oriented API of DEX is that it presents a much simpler interface to the user, and removes the need for a registry object. In reality, the registry object was performing a simplified subset of the object manager's tasks, such as handling instantiation of new objects.

The IUE provides an object manager to deal with IUE objects. To get an idea of how our mini-solution to the distributed object problem works with a larger system, this object manager was employed to describe IUE objects. The task of developing the program was made much easier, since the object manager handles the task of describing the objects in the DEX format, and also instantiates the objects from a DEX table. This description is bound to a character stream, and passed to another process using MPI, just as in our example. The object manager reads in the received stream, and creates an instance of the appropriate class, fills its state using the DEX description, and returns a pointer to it. The pointer returned is a void *, for the same reasons as in our system. This is casted to the type of the target object, and the replica is ready to use.

3.8 Further thoughts

To make this process completely transparent to the client, we could impose a layer that handles all invocations in the system. Local invocations can be routed to the appropriate object. In the case of a remote execution, the method we outlined above could be adopted to give the client the desired result. As we shall see in the next part of this study, this is roughly the method adopted in well known distributed architectures. The fundamental problem in our approach lies in object identity. A replica of an object is not the original object. What happens when the target object is

modified after the client receives its local replica? This copy becomes outdated, and the method invocations performed on it could yield inaccurate results. Some form of synchronization could be adopted to solve this problem. Alternately, the target object could be polled occasionally to see if it has changed. Many possible methods exist to solve this problem, but it is the mark of a complicated puzzle. An efficient, robust distributed architecture requires many components and involves intricate techniques that need to be properly assessed.

Since MPI supports MPMD execution also, the client and object could be split over different programs. However, there would have to be some way for the class declarations to be visible to the client process, so that a replica could be instantiated in its own space. Without this as common information, the problem becomes much more formidable, and the solution becomes much more valuable! As we shall see later, the method used to solve this is to avoid making replicas as we have done, and use some form of interprocess communication.

Part II

CORBA

In the previous section, we used an message passing layer to communicate information between processes, and built upon this a skeleton of a distributed object system. This system allowed us to have objects in different processes communicate with one another, stream themselves into a different process, and then simulate a remote invocation using this local replica. This provided some important insights into what is required in a distributed object system. We saw that it is necessary to have an efficient method of crossing process boundaries, and resolving differences in address spaces. In addition, a middle management layer is required to make the distribution of objects transparent to the client.

In the next part of this study, we will examine a full-fledged distributed object architecture that has been produced by a consortium of over 750 companies, over many years of combined effort. With the foundation that was established earlier, this architecture is more easily understood and appreciated for the level of service it provides.

Chapter 4

Overview of the Common Object Request Broker Architecture (CORBA) [1]

The notion of distributed computing has existed for a few decades, and has been the vision of many a computer scientist. However, the real impetus for making this a reality only came with the relatively recent change in hardware economics. As computing power and memory become cheaper, there are less reasons to maintain a system with large centralized servers, and more of a push towards personal workstations that can perform the same tasks in a distributed, cost-efficient manner. At the advent of any technology, there are various groups and forces involved, each pursuing different approaches to finding a solution. This continues until the need for standardization arises, which gathers these different parties and strives to produce the best consensus.

[1] This chapter was based largely on material from [Sie96 , spe96 , OHE96]

4.1 The Object Management Group

Established in 1989, The Object Management Group (OMG) was formed to develop this consensus in the areas of distributed computing and object oriented applications. The OMG is a consortium that now consists of over 750 software vendors, developers and end users. The mission of this organization is

> to promote the theory and practice of object technology for the development of distributed computing systems. The goal is to provide a common architectural framework for object oriented applications based on widely available interface specifications. [webb]

This framework is represented in the Object Management Architecture (OMA), shown in figure 4.1. The four components are

- **Object Request Broker (ORB):** A common communications bus for objects in a system to communicate. OMG's specification for the orb is called the Common Object Request Broker Architecture (CORBA).
- **Object Services:** (Also called CORBAServices) This provides the basic infrastructure and functions that almost any object would need, such as life cycle services, naming and directory services and others.
- **Common Facilities:** (Also called CORBAFacilities) This provides services to applications instead of objects, so it is at a higher layer of abstraction. Being a step towards application integration, this part of the OMA is further split into two parts:
  - **Horizontal Services:** those that can be used by almost any application, such as compound document services, help services and system administration.
  - **Vertical Services:** those that are particular to a certain industry, such as healthcare, telecommunications, banking, etc.
- **Application Objects** This is where independently developed application interfaces would fit in, although this has not been given too much attention by the OMG as yet.

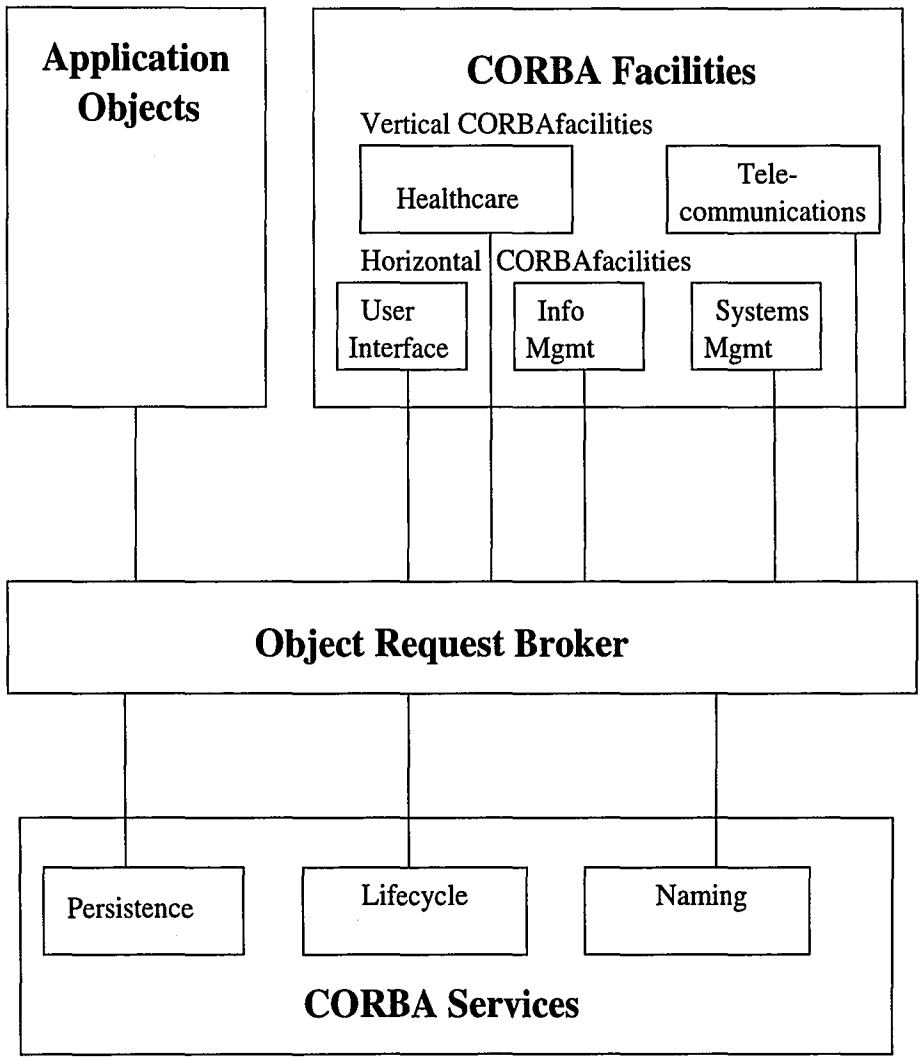

Figure 4.1: OMG's Object Management Architecture [Sie96]

As is evident, OMG has specified an entire architecture for object oriented applications, that penetrates up to the applications themselves. CORBA is only the bus for achieving transparent communication within this architecture, and according to OMG, the CORBAfacilities will eventually dwarf the other components. This effort will largely be undertaken by industry specific groups which will fulfill the specifications outlined by the OMG. The focus of this study being the actual construction and design of a distributed object environment, we shall only look at the technical aspects of the CORBA specification.

4.2 Basic Operation

Since the ORB is abstractly defined as a bus for communicating, the next natural question is "What uses this bus?" CORBA was designed to facilitate communication in an object oriented environment. Objects offer services as methods that can be requested, or invoked. The invoking party is referred to as a client. Naturally, it follows that a client makes an invocation on an object implementation over the ORB. Figure 4.2 depicts this. [2] One of the advantages of such a general framework is that it allows the client and object to be on different platforms and use different programming languages. To resolve these differences, a common medium is needed. The layer that separates the ORB from both the client and the object implementation provides this commonality.

4.2.1 Interface Definition Language

The methods that an object supports is collectively called its interface, since this is what the client uses to communicate with it. This intervening layer is thus called the Interface Definition Language (IDL). Although the syntax is derived largely from C++, IDL is a programming-language neutral way of expressing the services that an object provides. Being only a declarative language, there are no control constructs. It is only capable of describing objects and their interfaces. It is this simple feature that makes CORBA operable over various platforms and

[2]The normal terminology is client / server. However, it is common for a server to contain many objects. A client only makes requests from one object at a time. Therefore the term object implementation is more accurate than server in this context.

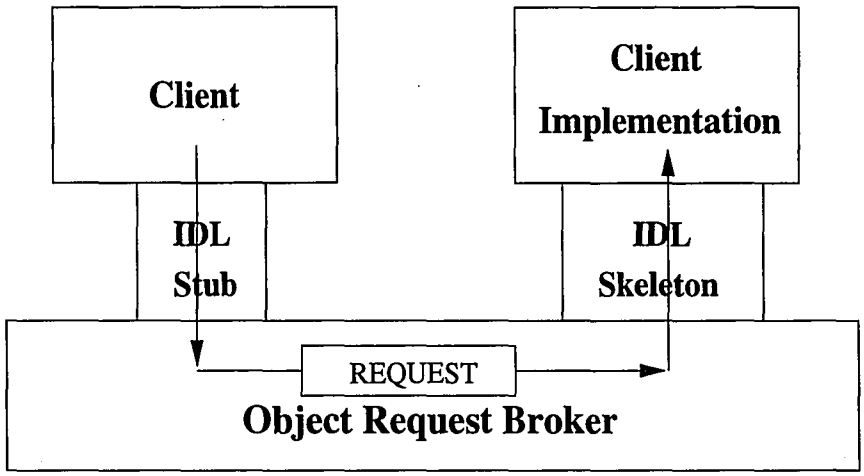

Figure 4.2: Client Object communication [Sie96]

environments. Regardless of what language the object is implemented in, its interface is described in IDL. The client can also use any programming language, but refers to the same IDL definition to perform a request. IDL also provides a layer of encapsulation, by only describing the interface of an object. Nothing is known about the implementation - how it is done, where it is located, or what it looks like.

Because of this encapsulation, anything in the implementation can change, or the object can itself be moved around the system. As long as its interface is the same, the client interacts with it in the same way. This gives tremendous flexibility to the object implementation, as well as a degree of robustness to the client program, which is very important in a dynamically changing software environment. To the client, the interface represents a promise. Given the parameters, the method will be executed, and if a return value is indicated, the client will receive the return value. The object views the interface as an obligation, which it must fulfill in any way. As we shall see, this method of encapsulation provides many advantages. Vendors of CORBA ORBs provide IDL compilers as part of their products.

An simple example of IDL can be seen in figure 4.3. It is apparent that this is a strongly typed language. Every parameter's type must be declared, as well as its usage - either in, out or inout. OMG's IDL also supports exceptions, which can be raised by the object implementation, and transferred by the ORB to the client, which deals with it appropriately. IDL interfaces can also

```
interface Person {
void SetAge (in short num);
}
```

Figure 4.3: An IDL interface

be inherited from other interfaces, providing a way to describe an object hierarchy.

4.2.2 Language mappings

The bridge between IDL and the host programming language is given in the form of a language mapping, which is implemented by the provider of the ORB. It is this mapping that specifies how IDL types, method invocations, and other constructs convert into actual function calls in the programming language. This language mapping is what presents the client with the view of the objects and the ORB interfaces.

Being defined in the CORBA 2.0 standard, these mappings have to adhere to certain rules. This ensures that IDL compilers from different vendors will generate the same functions and declarations that are needed to describe the object's interface. It is very rare that two IDL compliers produce the same program code, causing incompatibility between ORBs. However, this problem is solved by OMG, as we shall later see. However, an IDL compiler from one vendor might not produce the same host language code as another vendor's compiler, given the same IDL script, due to proprietary extensions and additional features. There are language mappings currently defined for C, C++, Smalltalk and Java.

4.2.3 Stubs and Skeletons

When an IDL file is passed through an IDL compiler, it uses this language mapping to generate code in the host language of the client and the object implementation respectively.

Figure 4.2 shows that the IDL layer has different names at the client and object implementation side. The client's interface to the ORB is called a **stub**, and is generated by the IDL compiler. This stub is then linked with the client program, and bridges the client with the ORB. The server

side has an IDL **skeleton** which bridges the object implementation with the ORB. Each object type needs a skeleton to bind to the ORB.

Figure 4.4 shows how the same IDL file is used by both client and the object implementation. The link between IDL and the client / object implementation is specified by the language mapping. However, the link between IDL and the ORB is left up to the ORB vendor to implement. It is therefore necessary to use the proper IDL compiler with the proper ORB. They are normally provided in pairs by vendors.

We have seen how the ORB is connected to client and object implementation programs, and have drawn a path connecting the two ends. There are important differences between the client-side and server-side architectures that are not evident in this view. To express these differences, it is necessary to take a closer look at the ORB itself.

4.3 Operation In Depth

We saw how IDL stubs connect the client to the ORB, and how skeletons connect the object implementation. An important aspect of these components is that they are statically linked with the client and object implementation respectively. With these elements as the sole link in our system, a change to any interface or the addition of an interface would require re-compilation of both the client and object implementation code. To fit our prerequisite of a dynamically evolving distributed system, we need something more, which will allow new objects to be added into the system. Clients should be able immediately access these new objects.

Figure 4.5 contains a more complete depiction of the various components involved in an ORB. From this it is apparent that the static stubs and skeletons are only one possible path from client to object implementation. Also seen in this figure are some new elements, which allow much more flexibility in the system.

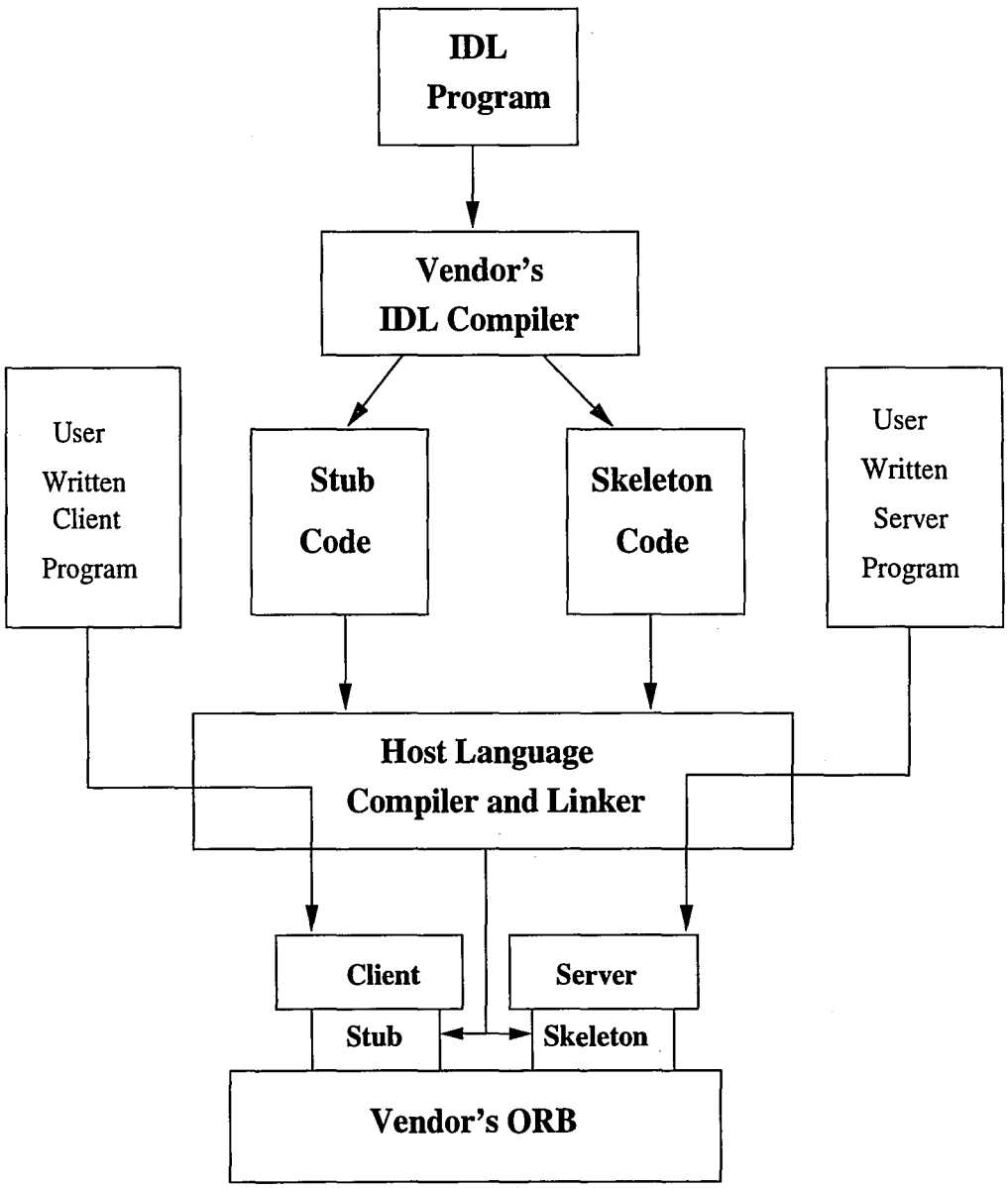

Figure 4.4: IDL file to client and object [Sie96]

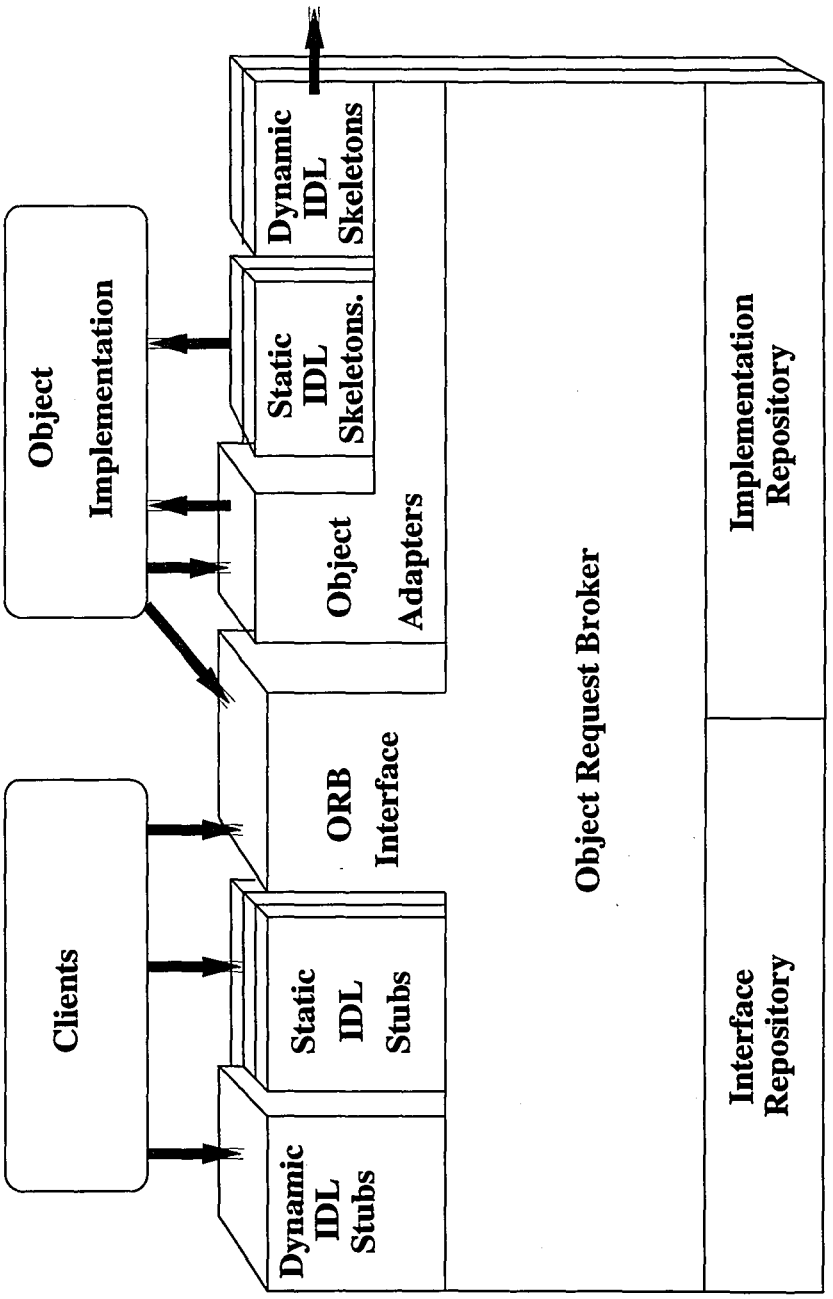

Figure 4.5: Structure of the ORB [Sie96]

4.3.1 The Interface Repository

CORBA requires that each ORB store the IDL information for all of its objects in the Interface Repository (IR). This is vital to the functioning of the entire system, and must be available to the ORB, the client and the object implementation. Many other utilities, such as object hierarchy browsers and debuggers also use this repository. The IDL interfaces can be added, deleted, modified, or retrieved from the IR. Using this to discover type information of parameters and signatures of functions, the IR can be used for interoperability between ORBS, and other purposes as well. A particular IR may be shared by multiple ORBS, and a single ORB may access more than one IR. The only requisite is that each ORB be able to access at least one IR. The CORBA specification does not indicate that the actual IDL code needs to be stored, although many IR's do precisely this. The information content of the interfaces must be preserved and be presentable to any query or search.

4.3.2 The Dynamic Invocation Interface

This uses the IR and allows the client to immediately make use of new objects in the system without having to be recompiled. It allows clients at run-time to:

- discover new objects
- discover their interfaces
- retrieve their interface definitions
- construct and dispatch invocations
- receive the resulting response or exception value

As seen, this allows the dynamic construction of object invocations, and can be used instead of calling a stub routine for a specific object. Although this method requires a sequence of calls to retrieve the proper interface, build and dispatch the call, this provides for a smaller executable image in memory, since it avoids the creation of a separate stub for each method of each object. A disadvantage is that use of the DII will not allow for type checking at compile time, which is

done when static stubs are used. Another major difference lies in the fact that DII invocations can be either in synchronous, asynchronous or deferred synchronous modes. Static invocations are generally synchronous, meaning that they block until the call completes. Lastly, both these methods allow dynamic binding, which is the selection of the appropriate object *instance* at runtime. Only DII allows the selection of object *type* and *operation* at runtime.

An judicious mix of static stubs and use of the DII can result in an efficient client. The DII is a good way to generalize client calls, and make the appropriate calls determining upon run-time conditions. The CORBA-2 specification indicates that "the nature of the dynamic invocation interface may vary substantially from one programming language mapping to another."

This shows what happens at the client side. The sole purpose of the client is to initiate requests, which may be passed to the ORB either through the static invocation interface (SII) using stubs or the dynamic invocation interface (DII). At the object implementation end, however, matters are more involved, as we shall see.

4.3.3 Implementation Repository

Although this repository is not covered in detail in the CORBA specification, there is a mention to its purpose. It is meant to contain information that will allow ORBS to locate and activate implementations of objects. This will naturally be specific to each particular ORB, and will vary in nature. The CORBA 2 specification also states that this will be "a common place to store additional information associated with the implementation of ORB objects." This could include debugging, administrative control, resource allocation, security, etc. [spe96]

4.3.4 Object Adapters

Before an object implementation is used, the actual object providing the service must be created, if it does not exist. If it already exists, it must be located. These services can be performed in many different ways, depending upon the type of the object. For instance, it could be created in the same process, or a different process. This decision could depend upon system resources, or even program requirements. Some objects might be in persistent storage such as a database.

Putting all this functionality into the ORB, which is the heart of this distributed system, would make it quite unmanageable.

To handle the diversity in the types of objects and their environments, CORBA uses an object adapter to handle the tasks of:

- registering implementations
- generating and interpreting object references
- mapping object references to their corresponding implementations
- activating and deactivating object implementations
- invoking methods
- coordinating interaction security

Object Adapters sit "on top" of the ORB and perform these services, accepting requests on behalf of object implementations. The object implementation in turn accesses most ORB services through an object adapter.

The OMG standard specifies that there should not be a large number of object adapters. It further outlines the specifications for a Basic Object Adapter (BOA) that can be used by most CORBA objects. The BOA has three different interfaces, which can be seen in figure 4.5:

- a private interface to the IDL skeleton
- a private interface to the ORB Core
- a public interface to the object implementation

To the BOA, a server is an execution unit, while an object implements a method or interface. Servers can contain many objects. The BOA supports four different activation policies:

- **Shared server policy:** A common process for all the instances of an object type

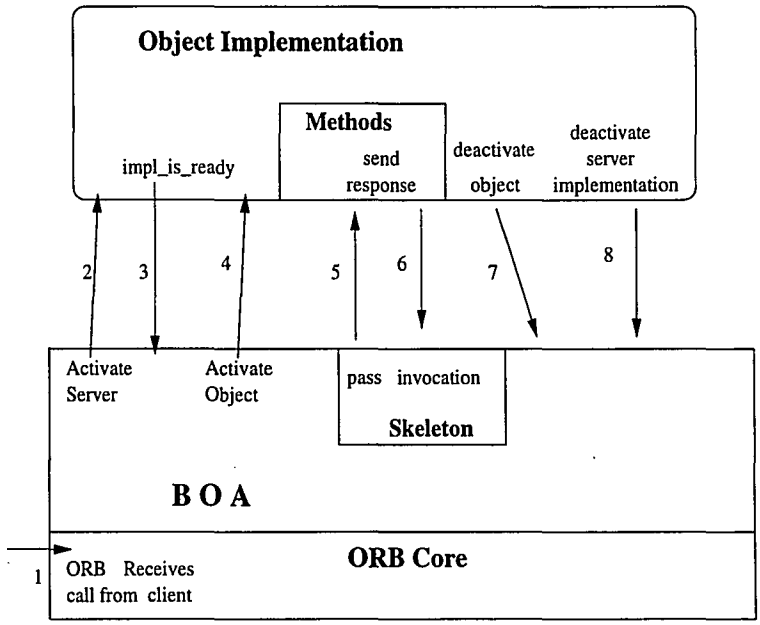

Figure 4.6: Steps on object implementation side [Sie96]

- **Persistent server policy:** Similar to the shared server, except that the server is activated outside the BOA and registered in an installation procedure
- **Unshared server policy:** One process per object
- **Server-per-method policy:** One process per method invocation

Each policy is appropriate for a different situation. Where process-initiation is expensive, a shared server would be preferred. Conversely, exclusive resources would require an unshared server. The server-per-method policy would be useful for load balancing.

4.3.5 Implementation Side events

Now that the various components on the object implementation end have been identified, we can trace the series of events commencing with the receipt of a request from the client by the ORB. Figure 4.6 portrays these steps.

1. The ORB receives a request. This includes information about the target object. The ORB checks its implementation repository and determines that neither server containing the object nor the object itself is currently active.

2. The ORB activates the server, passing it the information it needs to communicate with the BOA.

3. The server makes a BOA call, informing it that the server is ready to activate objects.

4. The BOA calls the server's object activation routine for the target object, passing it the object reference. The server activates the object. If this was previously active and made persistent, then its previous state is restored.

5. The BOA passes the requested invocation to the object through the skeleton and receives the response. This response is routed back to the client.

6. Additional requests to the object are passed through the skeleton as in step 5. If calls for additional objects in the same server are received, steps 4 and 5 are performed for each new object.

7. The server might decide to shut down an object, either because of a user request, or because of an event in the server's environment. It deactivates the object by making a BOA call. The BOA will no longer route calls to the object without reactivating it first. If the object is persistent, it saves it state before shutting down.

8. The server itself might shut down, in which case it de-registers itself with the BOA, informing it that it no longer is available to activate objects. If another request for this server arrives, the process recommences at step 1.

4.4 Remote Communication between ORBS

Most monolithic programs that operate on a single machine undergo major changes when they have to be converted to operate across machine boundaries. Remote execution is made remarkably transparent with a distributed object architecture such as CORBA. A CORBA client does

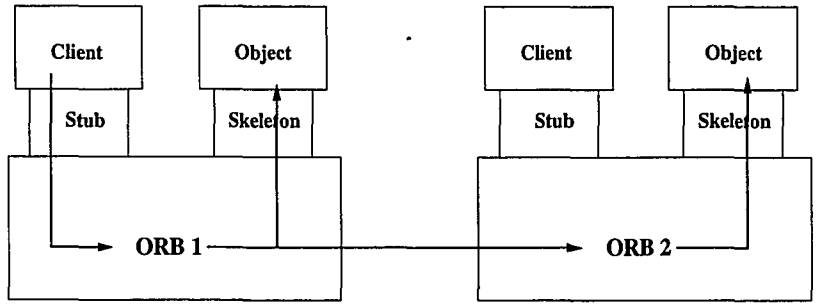

Figure 4.7: Local ORB can pass an invocation to another ORB [Sie96]

absolutely nothing different to pass a request to an object in the same process, a different process on the same machine, or even a different process on a different machine. A simplistic portrayal is shown in figure 4.7. This shows that the ORB simply passes the request to another ORB that supports the object desired, which could be on a different machine. This gives the client the illusion that the target object is in the same address space. The ORB actually searches its implementation repository to see if the implementation is local. If it is, then the request is passed through the skeleton. If it is remote, the request is passed across to the remote ORB, which routes it to the object. This entire process is transparent to the client, which makes a function call, and receives the response. This feature also allows objects on different types of ORBs to communicate, as we shall see.

Many things need to occur for the ORB to do this seamlessly. On the surface, an ORB must know where to send the request, and it must be able to consider network protocols, differences in byte ordering, etc. To achieve this communication, they must either recognize the same protocol, be able to translate each other's protocol, or there must be an intermediary which performs the translation. For an ORB to be CORBA 2 compliant, it must "speak" IIOP, or the Internet Inter-ORB Protocol. Based on TCP/IP, this protocol provides the common base for Inter-ORB communication. This will be examined in a later section. However, the local ORB must first have a way to form requests for object implementations that it knows nothing about.

4.4.1 Dynamic Skeleton Interface

This allows the dynamic handling of object requests, much like the DII on the client side. Static skeletons enable the ORB to pass requests to objects it knows about at compile time. However, in most cases, servers will not be given information about all the objects around the network. If the ORB does not know about a particular object at compile time, it needs a way to find it and pass requests to it at runtime. The DSI enables the ORB to dynamically form requests and pass requests to any implementation or proxy on the network. Actually two ORBS can construct a bridge that communicates the invocation to the remote ORB, and returns the response to the calling ORB. Naturally, this is possible only if the object implementation and the ORB support the DSI.

4.4.2 Interoperability

The CORBA specification includes a section on Interoperability, which is naturally important to any distributed system. It determines a standard that allows ORBs to support a network of objects that are distributed across and managed by multiple, maybe even heterogeneous ORBS. This is made possible by the concept of bridges, which specifies how ORBs that support different protocols can be connected. CORBA goes further to distinguish between immediate bridges and mediated, or half-bridges. Figure 4.8 shows this difference. Immediate bridges consist of a single bridge, which performs all the translation between the two ends. This method is fast and efficient when there are only two different types of ORBs. However, as the number of different ORBs grows, the number of bridges grows at the rate of $\frac{n2-n}{2}$, since each ORB needs a bridge to talk to all the other different types of ORBs.

To avoid this, half bridges can be used, in which there exists a backbone protocol. Each ORB talks to a bridge which translates to this backbone protocol. The number of bridges then only grows with the number of different types of ORBs at the rate of n. Disadvantages in this approach lie in the fact that messages need to be translated twice between the sender and destination, except when the destination ORB speaks the same protocol as the backbone. However, this layout matches the configuration of most large multi-protocol networks. Bridges can also be used to provide interoperability with non CORBA systems.

Figure 4.8: CORBA Bridges [Site96]

The General Inter ORB Protocol (GIOP) outlines a standard transfer syntax, or low level data representation, and a set of message formats for communications between ORBs. It is designed to work over any connection-oriented transport protocol, and can be mapped to a number of different transport protocols. The IIOP specifies how GIOP messages are exchanged over a TCP/IP connection. Being the most widely used transport protocol, this is the favorite candidate for backbone protocols between half-bridges.

Chapter 5

CORBA Implementation

After this overview of the features of CORBA, it is time to see an actual example of how programs are written using this system. It should be pointed out that developing software to fit the distributed computing paradigm is considerably different from single-system development. Entire textbooks have been written solely on the subject of distributed algorithms, which is only one part in the design of a system. In an distributed object oriented system, the distribution over multiple processes and machines might be transparent to the user, and even the application itself. However, this new environment is all but transparent to the developer, and must always be kept in mind, since it requires a different approach altogether. The purpose of this study is not to delve into the intricate aspects of developing a distributed application. We are more concerned with the architecture and design of the distributed system itself.

To understand the details of this design, and discover its strengths and weaknesses, a practical exposure to programming in CORBA is needed. It is well known that the biggest hurdle to cross in learning a language is to write one's first "Hello World" program. The first such program in CORBA shows us a lot about this system as well. First, let us digress a little and take a look at the various CORBA implementations available.

5.1 ORB Vendors

The OMG is a large consortium, with the lion's share of members being developers. It is a requirement within the OMG that once a company submits a proposal that is accepted within the consortium, the company must market a commercial implementation within a year [Sie96]. This helps keep proposals realistic, and brings out OMG's specifications as real products. This also means that there are many competing products that implement OMG's specifications. There are many different ORBs produced by various vendors. Each implements some or all of the CORBA specification, and each has their own proprietary extensions and additions.

The list of ORB vendors are widely varied, including major software houses such as SUN Microsystems and IBM as well as smaller organizations whose core competency is CORBA and middleware products. Many free ORBs are available for download, and a list of this is maintained on the Internet. [1]. Many ORBS support language mappings for one or more of the following: C, C++, Java, Smalltalk and Python, the most popular of these understandably being C++ and Java. The minimal requirement for an ORB to be CORBA-2 compliant is "adherence to the specifications in CORBA Core and one mapping." Interoperability is a separate compliance point. [spe96]

This study used two free ORBs to implement the basic tests. The first was MICO v0.9b6 (Mico Is COrba), developed at the University of Frankfurt in Germany. Having been developed for educational purposes, the entire source code was available under the Gnu copyright license. This provided some insight into the internals of CORBA, and allowed the tracing of invocations from client to object implementation. MICO supports:

- IDL to C++ mapping
- Dynamic Invocation Interface (DII)
- Dynamic Skeleton Interface (DSI)
- graphical Interface Repository browser that allows you to invoke arbitrary methods on arbitrary

[1] http://adams.patriot.net/ tvalesky/freecorba.html

- interfaces
- Interface Repository (IR)
- IIOP as native protocol (ORB prepared for multiprotocol support)
- support for nested method invocations
- full BOA implementation, including all activation modes, support for object migration and the
- implementation repository
- BOA can load object implementations into clients at runtime using loadable modules
- support for using MICO from within X11 applications (Xt and Qt)
- naming service

The second ORB which this study tried is OmniBroker version 2.01, developed by Object Oriented Concepts Inc. and offered free for non-commercial use. This too was available with the entire source code. The list of CORBA features that this ORB supports are:

- Full CORBA IDL support
- Complete CORBA IDL-to-C++ mapping
- Complete CORBA IDL-to-Java mapping
- Uses IIOP as native protocol
- Dynamic Invocation Interface
- Dynamic Skeleton Interface
- Interface Repository
- Peer-to-Peer communication with nested method invocations
- Support for non-blocking method invocations

- Support for timeouts
- Seamless integration with X11 and Windows
- A COS compliant Naming Service
- IDL-to-HTML translator for generating "javadoc"-style documentation

OmniBroker does not support different BOA activation modes, and can only activate objects in servers that have been launched manually.

5.2 Comparison of operation

To get an insight into the differences and similarities between different ORBs, we shall trace the process of creating a object and invoking its method using both MICO and OmniBroker. This reveals the steps involved in creating a usable CORBA object, as well as portability issues among CORBA products.

5.2.1 The IDL Definition

IDL is the fundamental building block of CORBA applications. Objects that are to work in the system must be defined by an IDL interface, and client programs rely solely upon this IDL definition to access these objects. Therefore, it is of value to see how different ORBS deal with IDL. The CORBA specification indicates **what** needs to be done, but says little of **how** to do it. Vendors are left to make their own design decisions, so long as the behavior is consistent with OMG's standard.

The sample IDL interface that was passed through IDL compilers of both ORBs is seen in 5.1. It describes a person object, which can have its age set, and can return its age using another method. OMG identifies strict lexical conventions and a grammar for IDL, making this standard across ORBs. IDL programs can be ported freely, and will compile with each ORB's IDL compiler. The disparity arises with what is done with this IDL. When an IDL program is compiled, the result is usually generated in the form of output files in the host programming language. This study focuses solely on the C++ mapping, and hence all examples are in this language.

```
interface Person {
void SetAge (in short num);
short GetAge ();
};
```

Figure 5.1: Simple IDL interface

5.2.2 Compiling the IDL

Using the C++ language mapping, the IDL compiler converts this interface definition into actual classes and source code that is used for coding the object and the client. The source files generated by this IDL script are given below:

Files generated by IDL compilers

MICO	**OmniBroker**
person.cc	person.cpp
person.h	person.h
	person_skel.cpp
	person_skel.h

It is clear that these ORBs do not do the same thing with the IDL script. MICO's ORB generates only one source and one header file while OmniBroker yields two of each. Figure 4.4 reveals what an IDL compiler must produce from an IDL file:

- stub code for the client
- skeleton code for the object implementation

Both ORBs meet these requirements, but in different ways. OmniBroker divides out the skeleton into a different file, so that the client program need not include the skeleton code. MICO combines both stub and skeleton into one file. A MICO client will still not know about the implementation details of a MICO object, since a skeleton stub is just an empty framework which must be filled in by the object. We shall now see how this is done.

5.2.3 The Object Implementation Side

After compiling the IDL interface definition, one of the components we are left with is a skeleton class that helps the ORB interact with our object implementation.

Inherit from the skeleton class

To allow this interaction to happen, the object implementation that we write is inherited from this skeleton class. Figure 5.2 shows what Mico's skeleton class looks like, and Figure 5.3 contains OmniBroker's skeleton class. This shows that these classes both multiply inherit from two classes. The first is Person, an abstract base class that is created for each object in the system. The second class that they derive from is different in each case. OmniBroker's skeleton derives from *CORBA_Object_skel*, while MICO's skeleton derives from *MethodDispatcher*. This shows that each ORB connects to the skeletons in different ways, proving that the link between the ORB and the object implementation differs between vendors. OMG solves this problem by introducing a Portable Object Adapter (POA) that specifies how base classes and skeletons are to be written.

These differences exist below the level of the object implementation. Besides using different naming conventions however, it is apparent that the code of these two skeletons are quite similar. By definition, a skeleton class is implemented as an abstract base class in C++. It is also seen that both contain a dispatch function, which the ORB uses to send invocations to the object. This dispatch function knows of all the methods implemented by the object (from the IDL definition), and calls the appropriate method of the person object, depending on which request is passed up by the object adapter (as discussed in section 4.3.5).

Fill out the implementation

Inheriting from the skeleton class gives our object the ability to be called by the ORB, and interact with it. However, the methods that comprise the interface still have to be "filled out" in order for the object to be of any use. For our simple example, there are only two methods, GetAge() and SetAge(), which are shown in figure 5.4. The implementations are identical for both ORBs. It should be pointed out that this degree of commonality is quite rare between

```
class Person_skel :
  virtual public MethodDispatcher,
  virtual public Person
{
  public:
    Person_skel( const CORBA::BOA::ReferenceData &
          = CORBA::BOA::ReferenceData() );
    virtual ~Person_skel();
    Person_skel( CORBA::Object_ptr obj );
    virtual bool dispatch( CORBA::ServerRequest_ptr _req,
                  CORBA::Environment &_env );
};
```

Figure 5.2: skeleton class generated by MICO's IDL compiler

```
class Person_skel : virtual public Person,
                    virtual public CORBA_Object_skel
{
    static CORBA_ULong _ob_num_;

    Person_skel(const Person_skel&);
    void operator=(const Person_skel&);
protected:
    Person_skel() { }
    Person_skel(const char*);

public:
    Person_ptr _this() { return Person::_duplicate(this); }
    virtual CORBA_ULong _OB_incNumber() const;
    virtual OBDispatchStatus _OB_dispatch(const char*,
                                          OBFixSeq< CORBA_Octet >&,
                                          bool,
                                          CORBA_ULong,
                                          CORBA_ULong);
};
```

Figure 5.3: skeleton class generated by OmniBroker's IDL compiler

object implementations of different ORBs. Many vendors add their own features and extensions to OMG's specifications. Object implementors use these features to enhance performance, or increase capabilities of their objects, but thereby also limit its portability to different ORBs. To a large extent, it is the object implementation and the server which determine how easily an application can be used across ORBs.

Write the Server

There are many approaches to designing and writing the server, depending upon how objects in the system are accessed. The discrepancies between ORBs is more apparent when it comes to server programs. The listings for both servers are in figure 5.5. The first difference lies in the way the ORB and BOA are initialized. The MICO server passes in a string reference in the third argument to determine whether an instance of the ORB must be created, or whether the server should use anoter orb. This is a good feature, and allows for considerable flexibility in development, but affects portability. The second difference lies in the way the server treats the new object.

MICO allows servers to be launched during runtime, by the BOA. It creates the person object, and then informs the BOA that the implementation is ready to receive requests. The problem of identifying and locating the object is left to the ORB and the client to solve. OmniBroker, on the other hand, requires that the server be launched manually. The person object is created, and a reference to it is "stringified" using the *object_to_string()* function. This reference is then written to a file called *person.ref* which the client can read. Following this, the BOA is informed that the implementation is ready to accept requests. These are two different approaches to making the object available to the client, and although the MICO server seems cleaner, it has drawbacks which will become apparent later. It should be noted that MICO is also capable of using stringified references, while OmniBroker cannot imitate MICO's transparent activation.

5.2.4 The Client Side

The client developer has a relatively cleaner task. The client stub generated by the IDL compiler does everything that is necessary to route the client's request to the ORB, which processes it

MICO example:

```
class Person_impl : virtual public Person_skel {
  CORBA::Short age;
public:
  Person_impl ()
  {
    age=0;
  }
  void SetAge (CORBA::Short num)
  {
    cout << "\nIn SetAge";
    age += num;
  };
  CORBA::Short GetAge() {
    cout << "\nIn GetAge";
    return ++age;
  }
};
```

OmniBroker example:

```
class Person_impl : virtual public Person_skel {
  CORBA_Short age;
public:
Person_impl()
{
}
void SetAge(CORBA_Short num)
{
  cout << "\nIn GetAge";
  age += num;
}
CORBA_Short GetAge()
{
  cout << "\nIn GetAge";
  return ++age;
}
};
```

Figure 5.4: Filled out examples of object implementations

MICO example:

```
int main( int argc, char *argv[] )
{
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");

  Person_impl * per = new Person_impl;

  boa->impl_is_ready (CORBA::ImplementationDef::_nil());
  return 0;
}
```

OmniBroker example:

```
main(int argc, char * argv[], char * [])
{

  CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
  CORBA_BOA_var boa = orb->BOA_init(argc, argv);

  Person_var per = new Person_impl();
  CORBA_String_var str = orb->object_to_string(per);
  const char * refFile = "person.ref";

  ofstream out(refFile);
  out << str << endl;
  out.close();

  boa->impl_is_ready(CORBA_ImplementationDef::_nil());

  return 0;
}
```

Figure 5.5: Servers in MICO and OmniBroker

and sends it further. This makes the underlying ORB complexity and object details relatively transparent to the client program. The client program listings are shown in 5.6. It can be seen that the basic order of steps are the same in both cases:

1. The ORB is initialized.
2. The client gets the reference for the object it desires. (Of type CORBA_Object_var)
3. The reference is casted down to the object type.
4. The client is able to invoke the object's methods.

The first point to note is that MICO again uses its string identifier in the third argument when initializing the ORB. Secondly, the method in which the client acquires the object reference is different, which matches the way the object was created in the server. We shall cover this, but first the type of the object reference is important to note. The client always gets a reference of type **CORBA_Object_var**, and then casts it to the type which it desires. This is to generalize the process and allow the client to get references for any type of object. It is even possible that the client gets a reference of an object which it does not know about at compile time, through the DII, as discussed in section 4.3.2.

Clients in the two systems use different methods of obtaining object references. MICO uses a specialized feature that allows the client to get an object reference from a naming service. Called the MICO Binder, this maps [Address, RepositoryID] pairs to object references. A repository ID is a string that identifies a CORBA object, consisting of the name of the object, and a version number. In the example, *IDL:Person:1.0* is such an id. Addresses can be local or on the network, to allow access to remote objects. The bind call in the client shows how this repository id is passed to the binder to retrieve the object reference of the person object. Although this obviates the need for the server to explicitly save the object reference for the client, applications using this feature are incompatible with other CORBA implementations. The OmniBroker client reads the stringified object reference from the file created by the server, *person.ref*, and converts it back to an actual object reference. If clients and servers are separated over the network, there must be a way for the client to have access to this file.

MICO example:

```
int main (int argc, char *argv[] )
{

  // ORB initialization
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");
  assert (argc == 2);

  CORBA::Object_var obj = orb->bind ("IDL:Person:1.0", argv[1]);
  assert (!CORBA::is_nil (obj));

  Person_var client = Person::_narrow(obj);

  client->SetAge(5);

  cout << "\nThe age has been set to: " << client->GetAge();

  return 0;
}
```

OmniBroker example:

```
main(int argc, char * argv[])
{
  //ORB initialization
  CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
  const char* refFile = "person.ref";
  ifstream in(refFile);
  char s[1000];
  in >> s;
  CORBA_Object_var obj = orb->string_to_object(s);

  assert (!CORBA_is_nil(obj));

  Person_var per = Person::_narrow(obj);
  assert (!CORBA_is_nil(per));
  per->SetAge(5);
  cout << "\nPerson's age is set to: " << per->GetAge() << endl;
  return 0;

}
```

Figure 5.6: Clients in MICO and OmniBroker

5.2.5 Running implementation

Although most issues in assessing these two ORBs arise in the development phase, there are subtle yet important differences in the way the final executables are run. To run the OmniBroker example, the server has to be started first, and then the client can be executed after that. Remember that the client must be able to read the file containing the stringified object reference. This could mean copying the file into the client's directory, if the location is not hard-coded into the client. If the client is running on a different machine, parameters indicating the host of the server can be passed in, which are given to the ORB at initialization time. The client can then issue a request which is passed to the server over the network.

MICO allows a similar situation, in which the server is started up first, and the client after that. In addition, the BOA can be started as a daemon process, called *micod*. The server must then be registered with the implementation repository, which is done using a command from the shell. After this, the client can be activated, and the BOA will automatically activate the server if it is not running. Following this, requests are passed to it in the normal way. This automation of activating the server can be very useful in systems where there are a large number of servers, which makes it difficult and inefficient to start each one individually. In this case, the server is only started when it is needed. The overhead lies in having to start the BOA daemon, and registering the server with the implementation repository. The entry for the server in the implementation repository indicates

- A name for the entry
- The mode in which the BOA should activate the server. This can be either
  - persistent
  - shared
  - unshared
  - per-method
  - library
- The command line to execute it the server

- The repository id's of the objects that the server implements.

All different modes were covered in section 4.3.4, except for the library mode. This allows the object implementation to be loaded as a module onto the running client. This is much faster than using Remote Procedure Calls (RPC) over different machines, and Inter-Process Communication (IPC) calls on the same machine. With this mode, the call would be as fast as a C++ method invocation. To allow this, the object implementation code would have to be changed, raising issues of portability again. This mode is not part of the CORBA 2.0 specification, and is a MICO extension.

Part III

COM/DCOM

After presenting CORBA, the base of OMG's framework for a distributed architecture, our study will now shift to another widely adopted specification developed by Microsoft. The Component Object Model (COM) was developed first, as new approach to object oriented software development. The Distribted Component Object Model (DCOM) was developed to extend this object model to operate over a network.

CORBA and DCOM are the two most widely accepted distributed object architectures. However, both take different approaches to our fundamental question of how to build such an architecture. After examining Microsoft's approach to this puzzle, we will be in a position to compare and contrast the two systems, and gain further insight into our purpose.

Chapter 6

Overview of the Component Object Model (COM) [1]

We have discussed some of the pitfalls of monolithic applications earlier, and the inherent problems they present to both developers and end users. To realize and overcome these pitfalls, the pioneers in this field borrowed key principles from many other arenas, including manufacturing. Any complicated piece of machinery consists of a number of parts. The utility and value of the machine would be greatly diminished if the entire thing would have to be scrapped because of the failure of a single part. Similarly, if a part was improved to increase performance, it would be useless to have to replace the entire unit. The founders of modern manufacturing solved this problem by splitting the unit into distinct components that could be individually fitted, removed, or upgraded. The monolithic approach to software development has exactly these problems.

From the phrase *Component Object Model* itself, a different way of conceptualizing software is suggested. Instead of building an application as a single static entity, this approach enables individual components to dynamically fit together and cooperate as a single application. The benefits of this are apparent in our example of the machine above. As we shall see, this forms a robust, extensible foundation for object oriented development. By extending this paradigm

[1] This chapter was based largely on material from [Rog97], [RK], [art96a]

to a distributed system, which is the focus of the next chapter, COM presents an efficient and practical method that fulfills our purpose. The original reason for developing COM however was "so software manufacturers could plug new accessories into existing applications without requiring a rebuild of the existing application." [RK] We shall see what makes this possible.

6.1 The Basics

6.1.1 Components

This framework developed by Microsoft revolves around a unit called the component, and basically specifies how they interact. A component is described as a miniature application, or a bundle of binary code that is compiled and linked, and is ready to use. This component can dynamically connect to other components at runtime through predefined interfaces, to form a whole application. The system can be changed by removing, adding, or upgrading these components.

6.1.2 Interfaces

It has been observed that an object is used by the methods that it implements. Similarly, a component is accessed by the set of abilities, or functions that it supports. A set of these functions comprise an **interface**, which is the sole view presented to clients of the component. For this reason, the interface is the fundamental building block of COM components. A component may support many interfaces. A component's interfaces also determine its behavior and interaction with other components, and hence constitute the behavior of the entire application. Because the components are encapsulated by their interfaces, they are free to change their implementation without affecting other components or the application.

Virtual Tables

In reality, an interface simply defines a memory layout for a group of virtual functions. Figure 6.1 shows what these tables, called Virtual Function Tables, or *vtbl's* look like. It is seen that

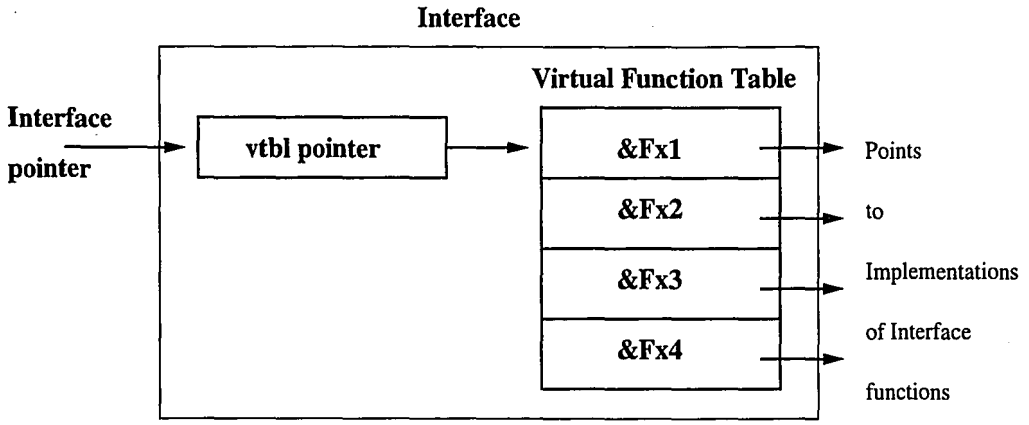

Figure 6.1: An Interface defines a memory layout for a group of virtual functions. [Rog97]

this structure is basically an array of pointers that point to the implementation of these virtual functions. This structure is in turn pointed to by a *vtbl pointer*, the use of which will be explained shortly. Since the only physical requirement of an interface is that it adhere to this memory layout, they can be implemented in any language that allows the creation of this structure, and the invocation of functions through this structure. This gives tremendous language flexibility, since this is essentially a specification at the binary level.

Interfaces in C++

In C++, abstract base classes provide the exact layout as seen in figure 6.1. Interfaces are represented as abstract base classes, and components can be represented as classes that inherit the interfaces it supports. A simple example is seen in figure 6.2. It is seen that a component can implement as many interfaces as it chooses. Since COM specifies a binary memory layout for interfaces, there is no formal requirement that a component be implemented using one class. It can use a different class for the implementation of each one of its interfaces, or the component may be represented without using classes at all. Because of the nature of C++, and its memory layout, this method is the most convenient, and is generally adopted.

An interface as an abstract base class

```
class MyInterface1
{
public:
  virtual void SomeFunction1(int a) = 0;
};
class MyInterface2
{
public:
  virtual void SomeFunction2(int * b) = 0;
};
```

A component inherits these interfaces

```
class MyComponent : public MyInterface1,
    public MyInterface2
{
  //Implementation of abstract base class MyInterface1
  virtual void SomeFunction1(int a)
  {
    cout << "\nFunction of MyInterface1";
  }
  //Implementation of abstract base class MyInterface2
  virtual void SomeFunction2(int * b)
  {
    cout << "\nFunction of MyInterface2";
  }
}
```

Figure 6.2: An example of an interface and a class in C++

Properties of Interfaces

Once an interface is developed and published, it cannot ever change. Instead of changing its behavior, and hence affecting all the components that support the interface, a new interface is created. Components that desire the changed behavior will simply start supporting the new interface, while the old one can still be used by those that need it. Allowing components to support multiple interfaces allows for this stability in new versions of interfaces. This feature also allows clients to treat components polymorphically. If two components support the same interface, clients can treat them with the same piece of code.

When designing interfaces, it should be noted that small interfaces, that specify a single behavior, allow for greater reuse of components than large interfaces that contain many behaviors. The more an interface tries to do, the more specific it becomes to a particular situation. Hence, the chances are less that another component would be able to use the same interface. Reusable architectures are developed by using well defined, small generic interfaces. This also allows greater degrees of polymorphism, by letting clients deal with different components in the same manner.

Vtbl Pointers

Figure 6.1 shows a pointer to the vtbl, which adds an extra layer of indirection to access the functions of the interface. This is done for many reasons. First, it allows the storage of instance specific data with the vtbl pointer, as in figure 6.3. While C++ classes can manipulate class instance data directly, COM components never have access to this data. Only the interface functions can use instance specific data, and hence only they need to know about its existence. Vtbl pointers also allow different instances of a class to share the same vtbl. The instance specific data for each class may vary, but the functions are common, which allows them to share the same vtbl.

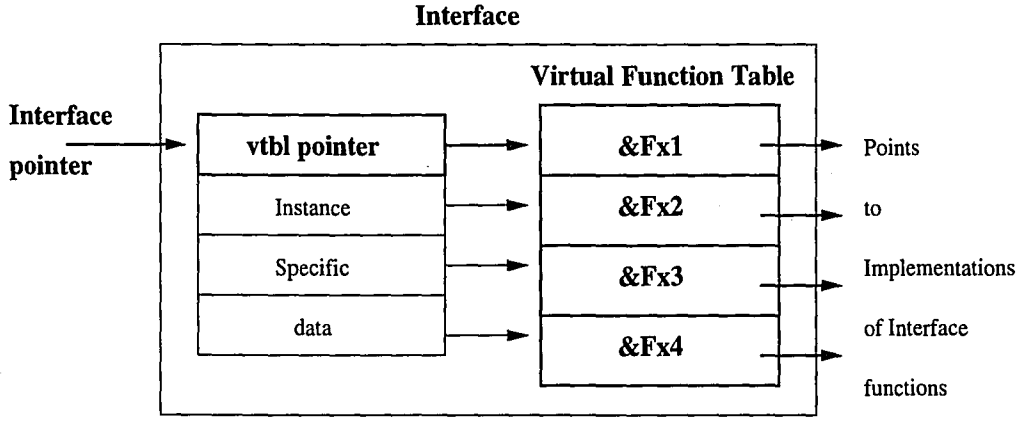

Figure 6.3: Instance specific data stored with Vtbl pointer [Rog97]

6.1.3 Unique Identifiers

In any reasonably sized system, there are sure to be a large number of components. This naturally means that there will be a large number of interfaces as well. To ensure that the correct interface and component are used for any operation, it is necessary that unique identifiers be assigned to each component and each interface. COM uses the method of Globally Unique Identifiers, or *GUID*'s to identify its components and interfaces. [2] Environment (DCE). GUID's are basically 128 bit integers that are algorithmically computed to be unique over space and time. [3] This allows people in different places to cooperate on a component or application without worrying about redundant component or interface identifiers. Human-readable names, which are less accurate, are used locally for readability. GUID's can be generated by using a Windows utility called GUIDGEN.EXE or UUIDGEN.EXE.

Interfaces are identified by GUIDs called interface identifiers *(IIDs)*. while components are identified by GUID's called class identifiers (CLSIDs). Although two components may implement the same set of interfaces, they will still have different CLSIDs. In addition, components can add interfaces without changing their CLSIDs, which does not apply to interfaces and their IIDs. These identifiers are built into the binary code of a component, and are used dynamically at

[2]GUIDs were defined by Open Software Foundation's (OSF) Distributed Computing, although they call them UUIDs. [Rog97]

[3]The algorithm is computed by using the address of the machine's network card, which makes it unique in space, as well as the time stamp, making it unique in time. The algorithm currently used will roll over in approximately AD 3400. [Rog97]

runtime to ensure proper connections between components, and the proper use of interfaces.

6.2 Using Components

We have described the building blocks of the COM architecture, namely components and the interfaces they implement. Now we shall determine how they are used in a system. A component implements one or more interfaces to express its behavior, and establish a contract with clients defining this behavior. Clients use this contract through the interfaces and employ the services of the component. In order for this to happen, the client must have a way of

- Finding out what services a component offers
- Referring to a component
- Using these services

6.2.1 The Unknown Interface

Since a component may implement a number of interfaces, which can vary across components and systems, there needs to be a standard way for a client to determine what services a component offers. This is made possible by requiring every component to implement an interface called *IUnknown*. A client acquires an pointer to the IUnknown interface of a component when it is created, which comes later. Despite its name, a client can be assured that a COM component implements this interface, consisting of three functions:

1. QueryInterface
2. AddRef
3. Release

In addition, every interface must inherit from IUnknown, making these functions the first three entries in their vtbl. This also allows all interfaces to be used polymorphically as IUnkown interfaces, which is important in many ways, as we shall see.

QueryInterface

QueryInterface is a function that allows clients to discover whether a component supports a particular interface. The signature of the function is

```
HRESULT __stdcall QueryInterface(const IID& iid, void ** ppv)
```

The first parameter is the interface id for which the client is searching, and the second parameter is used to return the pointer to the interface, if the component supports it. Since every interface inherits from IUnknown, the client can query any interface of a component, and find out if an interface is supported. This ability to query any interface obviates the need to keep track of components. These interface pointers are the only thing the client uses to communicate with a component. Since the QueryInterface method is so vital in the communication between client and component, there are some standard rules regarding its behavior:

- *All interfaces in a component will return the same IUnknown interface.* Each instance of a component has only one implementation of an IUnkown interface, so no matter which interface of the component is queried for IUnknown, the same pointer is returned.
- *An interface can be acquired at any time if it exists.* If QueryInterface succeeds for a given interface of an instance of a component, it will always succeed. Similarly, if it fails for this instance, it will always fail. This may change for a new instance of the component.
- *The currently held interface can be queried for.* If a client has a pointer for a particular interface, it can query that interface for itself and get the same interface back. Since all interfaces inherit from IUnknown, this rule can be used to get any interface from an IUnknown pointer, by the principle of polymorphism.
- *An interface pointer received by querying an interface of a component can be received from any interface of the component.* This allows a client to query any interface of a component, and get the same response regardless of which interface it is.

QueryInterface is what defines a component, in that it is the only way for a client to know what interfaces are supported. This makes it a fundamentally important part of the COM component. The other two methods of IUnkown deal with controlling the life-cycle of a component.

6.2.2 Reference Counting

Life-cycles of components need to be controlled, especially in large systems where resources must be wisely allocated. In our scenario of a COM system, components are accessed by clients, who use their services through the interfaces that are supported. In a one-on-one case, it is simple to know when a client is using a component, and when it is finished with it. However, since components can support many interfaces, it is possible for clients to be using more than one interface of a component at a given time. That is, a client may hold the pointers to more than one interface, and require both interfaces of the component for its task. In complicated cases, where there are many clients accessing many interfaces on many components, it is quite difficult to assess when a component is no longer needed. In this case, there needs to be a method of determining when a component can be safely deactivated.

Since the multiplicity exists on the client side, the control for determining when to deactivate a component must rest with the component itself. This is made possible by a simple technique called reference counting, which requires the maintenance of a counter by the component. When a client gets an interface from a component, the component's reference count is incremented, using the *Addref* method of the IUnknown interface. When the client is done with the interface, it releases it using *Release*, and the component decreases its reference count. When this count drops to zero, the component is no longer being used, and it can deactivate itself safely. The component is the central point for its clients, and can maintain this count for the component as a whole, or one for each interface it supports. In either case, the component can decide when it is no longer needed, and delete itself.

6.3 Creating a component

It is now possible for a client to find out which interfaces a component supports, and acquire a pointer to the interface it desires. We also have a way of controlling the life-cycle of components to maintain stability in our system. However, a more fundamental issue arises of creating the component. COM gives this responsibility to the client, since it knows best when it needs a

component, and which one it needs. This is done by a *CoCreateInstance* call, which takes the following parameters:

```
HRESULT __stdcall CoCreateInstance(
const CLSID & clsid,
IUnknown * pUnknownOuter //Used for Aggregation
DWORD dwClsContext,      //Server Context
const IID& iid,
void ** ppv
);
```

The first parameter indicates the CLSID of the component to be created. The second parameter is used in cases of aggregation, which will be covered later. Third is the parameter which restricts the execution context of the component to be created. The fourth parameter specifies the IID of the interface the client wishes to use. By specifying this, the client need not use QueryInterface immediately after creating a component. The fifth parameter returns the pointer to the interface which the client requested. Components may run in the same process as the client, in a different process, or on a different machine. The third parameter *dwClsContext* controls this, and can be a combination of the following flags:

- **CLSCTX_INPROC_SERVER** The client will accept components that run in the same process. Components must be implemented in DLLs to run in the same process.
- **CLSCTX_INPROC_HANDLER** The client will use in-proc handlers, which are in-process components that implement only part of a component. The other parts of the component are implemented by an out-of-process component on a local or remote server.
- **CLSCTX_LOCAL_SERVER** The client will use components that run in a different process but on the same machine. These servers are implemented as executable programs.
- **CLSCTX_REMOTE_SERVER** The client will accept components that run on different machines. This flag requires Distributed COM, which will be covered in the next chapter.

Components can be available in all three contexts - in-process, local and remote. It will depend upon the application and the client how they are executed. This introduces the idea of controlling how components are created. This chapter will focuses on COM, covering components that are in the same process as the client, and those in a different process on the same machine. DCOM is about components that can be separated from the client over a network.

6.3.1 In Process Components

We have established that a client accesses a component through its interface, which is simply a table of pointers to its functions. The memory these pointers dereference must be accessible to the client in order for it to use the functions of a component. The easiest way to do this is to implement both client and component in the same address space. An executing program is known as a process, which has its own address space in memory, distinct from other processes. [4] Pointers in one application cannot be passed to another application, since their address spaces are different. The value of the pointers may be the same, but the addresses they point to would be different.

There are two ways that components can be implemented in the same process as the client. First, they can be statically linked, and made part of the same application. This contradicts the very purpose of COM, since a change to either client or component would require re-compilation of the entire application. A more elegant way of achieving the same result is to implement the component in a dynamic link library (DLL), which is loaded into the client's address space at runtime. This separates the client's and component's implementation, but allows them to run in the same process. Hence, a vtbl of function pointers can be supported by a component and meaningfully used by the client. More than one component may be implemented in a DLL, and the client can create any one it desires.

[4] In Windows, each process has 4 GigaBytes for its address space. [Rog97]

6.3.2 Local Process Components

There are situations that require components to be implemented in their own process, as an executable piece of code. DLLs have to be linked against client programs, to allow the clients to access the components, and this might not always be possible, especially in large development situations spread over vast distances and times. Executables can be developed independently, and by publishing the interfaces, the client and component programs can come together without any prior knowledge of each other. However, this entails the crossing of process boundaries, and communication over different address spaces, which affects our vtbls, the fundamental layout of a component's interfaces.

Figure 6.4 depicts this scenario. It is evident that the pointers to the functions of the component will have no meaning in the client's address space. To effectively deal with this problem, the following conditions must be satisfied:

- A process should be allowed to call a function in another process.
- Data should be accurately passed between processes.
- A client should not bother about whether the component it is accessing is in-process or out-of-process. This should be transparent to the client.

To communicate between processes, many methods have been developed, such as pipes and shared memory. However, COM uses local procedure calls (LPCs), which is based on the remote procedure call (RPC), as defined in OSF's DCE specification. LPCs allow communication between processes on the same machine, while RPCs operate over different machines. This method utilizes the fact that the underlying operating systems knows of every process's address space, and hence can call functions in any process. When a client acquires an interface of a component, a LPC channel is established, allowing the client and component to communicate as if they were in the same address space.

Now that a process can call a function in another process, the next step is to outline a scheme for passing data between processes, so that we can pass the parameters to these functions. This is accomplished by using a technique called *marshaling*, which considers many issues when copying

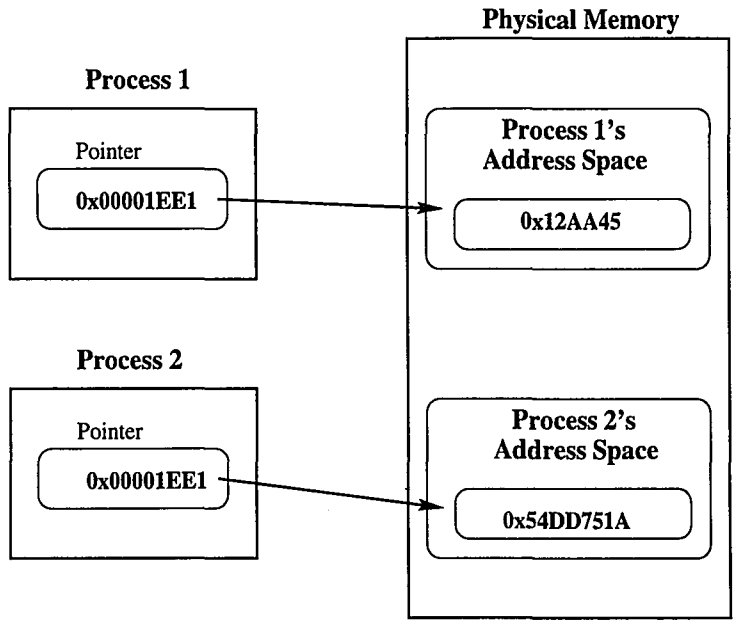

Figure 6.4: Out of process components have to deal with different address spaces [Rog97]

the data from one address space to another. For instance, basic data types, such as integers and chars can be copied in a straightforward manner, but pointers need to be dereferenced, and the memory that they point to should be copied. However, if the pointer is an interface pointer, then the memory it points to should not be copied, but the receiving process should be given a pointer that will allow access to this same interface. Clearly, marshaling is an important and intricate issue. This function can be performed by an interface called *IMarshal*, which COM queries a component for at creation time. By implementing a *IMarshal* interface, an object specifies what data is marshaled, and how it is done. Without this, COM uses its standard marshaling method.

To make this process transparent to the client, a call to an in-process component should be no different from a call to a local process component. This is made possible by adopting the method pictured in figure 6.5 It is seen that a client still communicates with a DLL, as in the in-process case. This DLL is called the *proxy*. Similarly, the component, which is in its own executable, also communicates with another DLL called the *stub*. The proxy marshals the parameters and passes them on to the stub, with any other relevant information, such as the interface and function desired. The stub unmarshals these parameters, makes the proper invocation on the desired component, and passes it the parameters. From the client's view, this is the same as

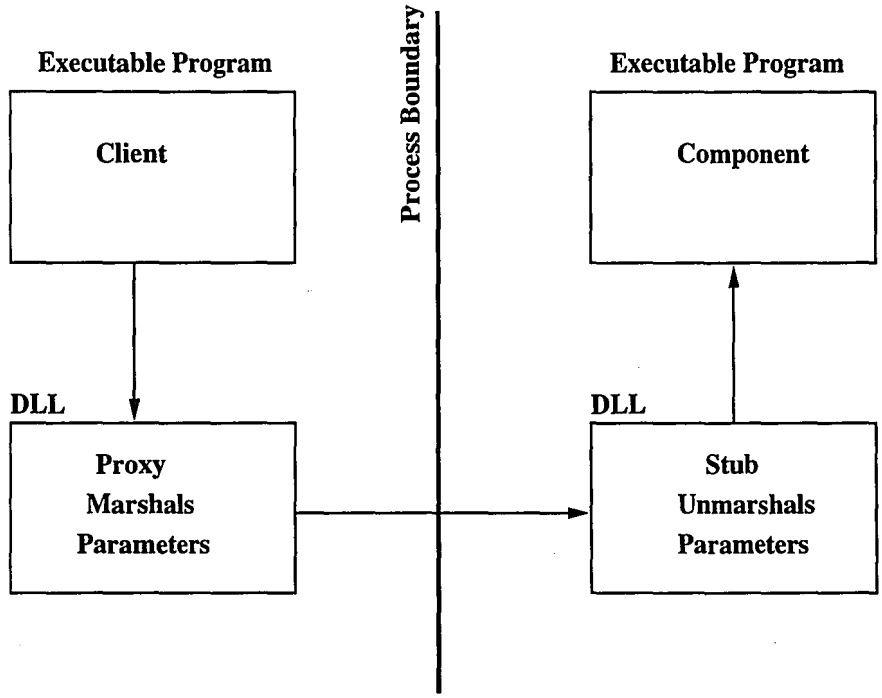

Figure 6.5: A client communicates with a DLL, which marshals the function parameters and calls the stub DLL using LPCs. The stub unmarshals parameters and makes calls on the component to which it is linked. [Rog97]

communicating with a component in the same process.

6.3.3 The Class Factory

As we have seen, it is possible to have components in the same process, implemented in a DLL, and in a different process, using the method outlined in the previous section. It is apparent that a component can be created with many options. However, once it is created by the client, it is too late to change these options. To create components in a flexible yet generic way, COM uses an intermediate agent called a class factory. This class factory is implemented by the component developer, and is responsible for creating all the instances of a particular component type. Class factories support the standard interface *IClassFactory*, which contains two member functions:

1. **CreateInstance** Responsible for instantiation of the component

2. **LockServer** Provides a way for the client to keep the server active in memory if it needs its services.

The *CoCreateInstance* call actually triggers the COM Library to create the class factory of the specified component, which creates an instance of the component using its CreateInstance function. After instantiating the component, the class factory invokes its QueryInterface method to get the desired interface. This interface pointer is then passed back to *CoCreateInstance* or the client who created the class factory. The client now has a pointer to the desired interface of the component. It is also possible to have a single class factory create different types of components, by giving it access to the creation functions of all these components. In addition, a server might not always create a new instance of the component. It could return an interface pointer of an existing component, so that clients can connect to the same instance with a particular state.

6.3.4 Interface Definition Language

Apparently, the creation of an out-of-process server requires more components than an in-process server. To write a proxy and stub for each component could be very time consuming. To alleviate this burden, COM provides a way to describe the interfaces of the component in a language called the Interface Definition Language (IDL). The MIDL compiler is then used to generate the proxy and stub DLLs, which can be linked with the client and component executable respectively. While it is not necessary to use IDL to develop COM components and clients, it releases the developer from low-level details of LPC calls. The IDL compiler can also generate a *type library* that stores information about the interfaces and their methods. This may be used at runtime for dynamic invocations, which allow clients to access components it does not know about at compile time. [CYY] A sample IDL description of the interface in figure 6.2 is given in figure 6.6. Note the use of HRESULTs, which are described in section 7.3.3. Also, parameters are specified to be in, out or inout, to optimize proxy and stub code. The MIDL compiler requires that all out and inout parameters be pointers.

An interface as an abstract base class

```
[
  object,
  uuid(5e9ddec7-5767-11cf-beab-00aa006c3606),
  helpstring("MyInterface1"),
  pointer_default(unique)
]
interface MyInterface1 : IUnknown
{
  HRESULT SomeFunction1([in]int a);
}

[
  object,
  uuid(5e9ddec8-5767-11cf-beab-00aa006c3606),
  helpstring("MyInterface2"),
  pointer_default(unique)
]
interface MyInterface2 : IUnknown
{
  HRESULT SomeFunction2([out] int *b);
}
```

Figure 6.6: IDL for our example interface

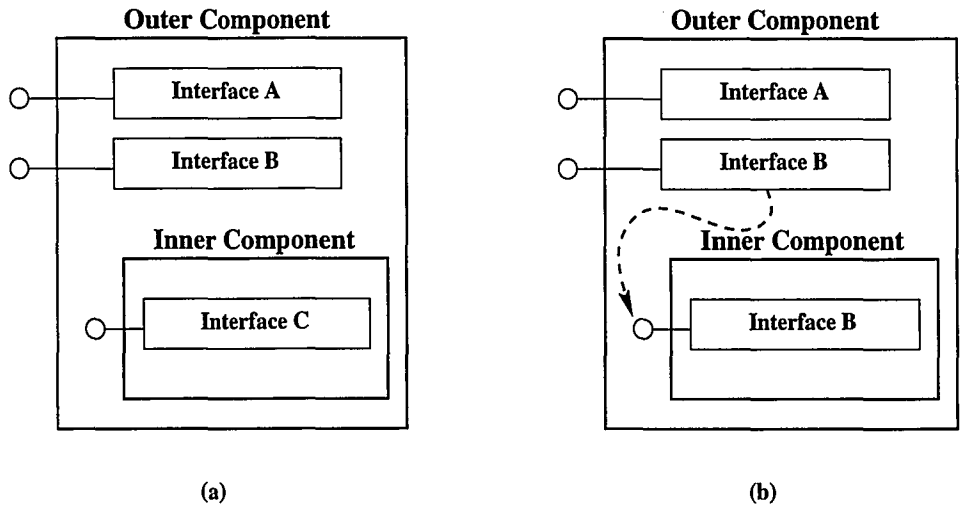

Figure 6.7: (a) Containment (b) Reimplementation of inner component's interface [Rog97]

Inside COM - page 161m 162

6.4 Aggregation and Containment

In object oriented development, inheritance is a necessary tool to express hierarchies of relation between objects. Traditional inheritance of C++ classes ordain that a child inherits **both** the implementation and the methods of a parent class. In large hierarchies, this could potentially cause problems when the implementation of a parent class is changed. Great care is necessary to ensure that this change does not break one of the children that depend on this particular implementation feature. To avoid this situation, COM only supports interface inheritance. This fits in with the fact that interfaces are the pivotal unit of COM.

However, the ability to inherit the implementation of a component can be useful in many situations to increase reuse and maintain consistency. COM simulates implementation inheritance by using the techniques of containment and aggregation. Containment is portrayed in figure 6.7(a) and is achieved by having the outer component maintain pointers to the interfaces of the inner component. In this method, the outer component acts as a client of the inner component. The interfaces of the inner component may be used by the outer component for its tasks, or they may be re-implemented to be presented to the outside client. This is done by forwarding calls to the inner component, as in figure 6.7(b).

Aggregation adopts a different approach. Instead of re-implementing an inner component's interface and forwarding client calls to it, it passes the inner component's interface to the client. From this point onwards, the client communicates directly with the inner component. An important point is that the client sees only one component that supports different interfaces. It is not aware of the aggregation on the component side. Enabling the outer and inner components to behave as one is achieved by the following conditions:

- The inner component has two IUnknown interfaces, as shown in figure 6.8.
- The inner component forwards all IUnknown calls to the outer component's IUnknown interface through the Delegating IUnknown implementation.
- The outer component maintains a private pointer to the inner component's Nondelegating IUnknown implementation.
- The outer component passes a pointer to its IUnknown interface to the inner component when it is created, using the second parameter of CoCreateInstance, as shown in section 6.3.

This presents a view of a single component to the client, and maintains the rules of using QueryInterface, as discussed in section 6.2.1. Using these methods of aggregation and containment, it is possible for COM components to simulate implementation inheritance, an important aspect of object oriented applications.

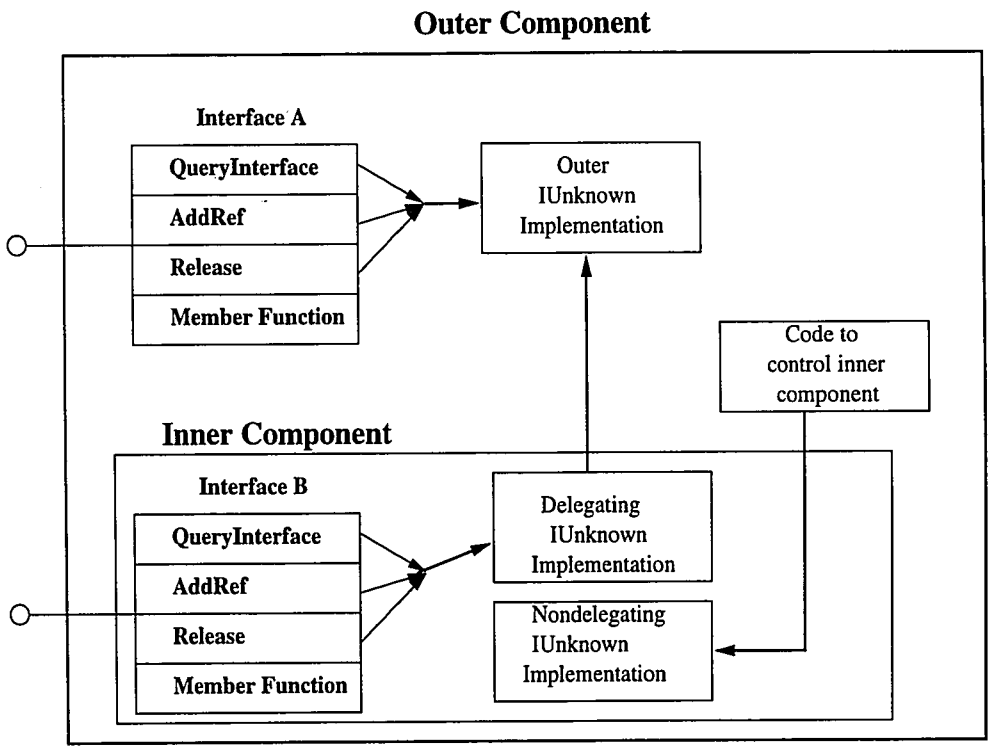

Figure 6.8: Aggregation in COM [Rog97]

Chapter 7

Overview of the Distributed COM (DCOM) [1]

The previous section covered Microsoft's Component Object Model, which presents an innovative approach to object oriented development using its methodology for the interaction between components in a system and their clients. Extending this model to operate over a network, the Distributed Component Object Model, or DCOM, outlines a comprehensive environment for a distributed object oriented system. In this chapter, we shall examine how DCOM extends the operation of COM, and what it additional features it offers.

7.1 COM over a longer wire

A large part of the groundwork for DCOM has already been done by COM, which is why an entire chapter was devoted to it. It was seen how COM enables clients and components to interact in the same process, and even across process boundaries. Figure 6.5 depicts this interaction between distinct processes. In the situation where the client and component are separated across different machines, DCOM enables them to communicate by replacing the LPC layer with a network

[1] This chapter was based largely on material from [art96b], [CYY]

protocol stack that enables remote communication between processes. This can be thought of as replacing LPCs with RPCs, although DCOM extends its network protocol layer beyond OSF's specification of RPC. Figure 7.1(a) shows the COM architecture, while 7.1(b) portrays the difference in DCOM.

It is apparent that this change is completely transparent to the client. From its view, it does not matter whether the component it is accessing is in the same process, or on a machine at the other side of the world! Once the client acquires an interface of the component, an RPC channel is established, allowing the client and component to communicate as if they are in the same address space. This allows for considerable flexibility in developing DCOM clients and components, and ease in modifying COM applications to fit the distributed paradigm.

From an architectural standpoint, this is the only change needed to distribute COM components. However, practical considerations of network characteristics, and the inherent complications of maintaining robust communication across different machines require some subtle changes to components and their clients, as is discussed below.

7.2 Factors affecting distributed applications

When communicating processes are on the same machine, the most formidable hurdle was that of operating across different address spaces. The underlying operating system and hardware is the same for both component and client, which allows them to completely ignore issues of lower compatibility. However, when communicating across a network, the amount of commonality is greatly diminished. Clients and components can be on different types of hardware architectures, use different operating systems, and face vastly different environments. Networks can be heterogeneous collections of machines, transport protocols, byte ordering, etc. Clearly, the jump from inter-process to inter-machine communication is not an easy one.

In addition, processes on a local machine are relatively safe from unforeseen termination. When the machine goes down, both client and component go down with it. In a distributed environment, a component that suddenly crashes could have damaging consequences for the clients accessing it. Conversely, client crashes also affect components, as we shall see. Network latency also

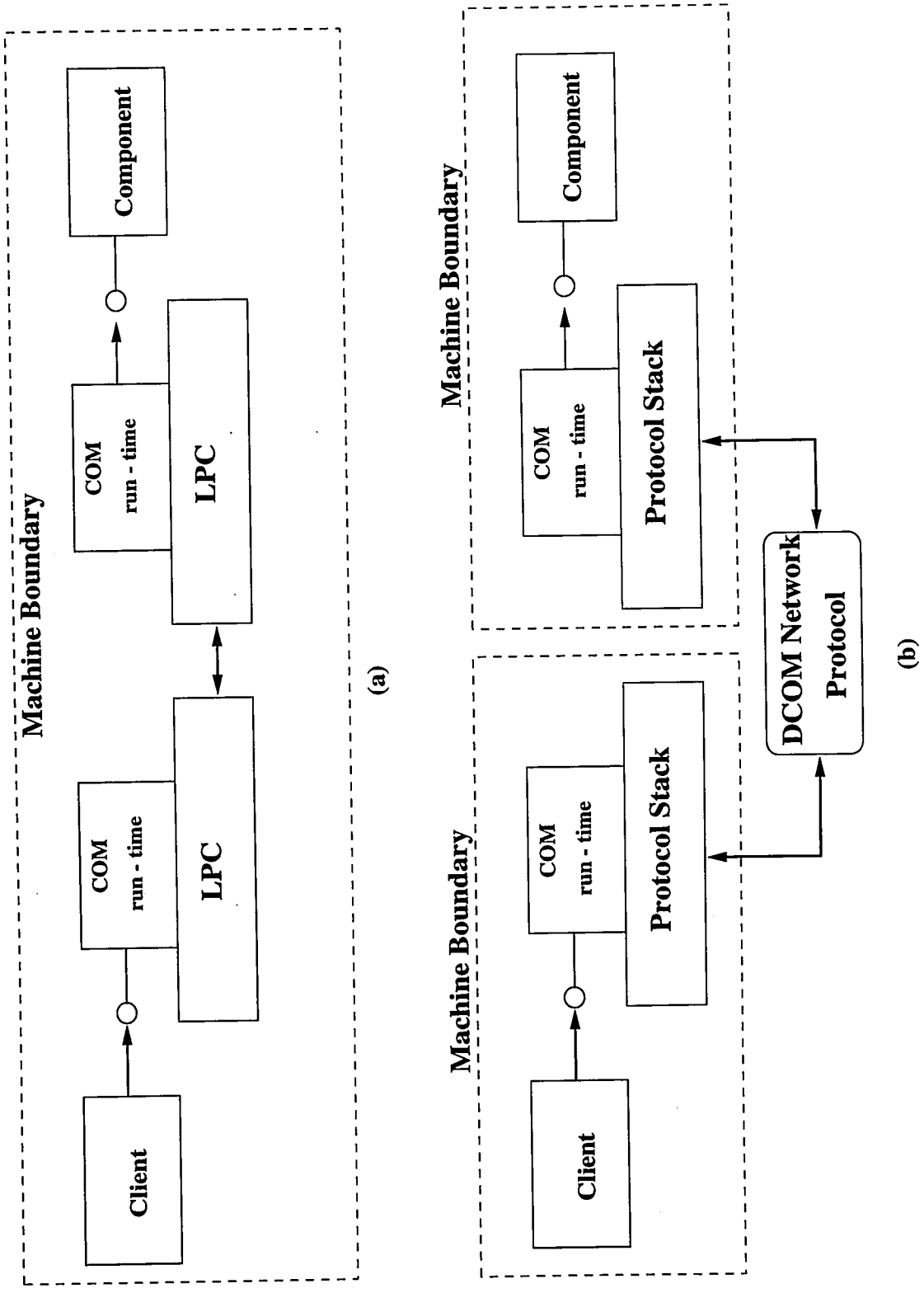

Figure 7.1: DCOM replaces LPC with RPC

raises issues of performance and timing, especially in Wide Area Networks and the Internet. To summarize, there are many realistic considerations that DCOM faces in its attempt to separate clients and components across a network.

7.3 Changes in client - component interaction

7.3.1 Pinging protocol

In order to deal with these factors, DCOM components are given some additional abilities. It is seen that the lifetime of a component is controlled by maintaining a reference count that indicates whether the component is in use or not. This method works well in the local scenario. However, consider the possibility of a client and component being on different machines. The client creates a component and acquires one of its interfaces, hence incrementing its reference count. In the course of operation, the client machine crashes, and the client is terminated before it lets go of the interface it had. The component's reference count will never be decremented to zero because of this single outstanding connection, which in reality no longer exists. Consequently, the component will not be able to remove itself. Extrapolated to a larger scale, and considering the likelihood of a machine or network failure, this is a potentially critical situation. This is handled by using a *pinging protocol* to detect whether clients are still active. Client machines send a periodic message, and the component considers a connection as broken if more than a predefined number of ping periods pass without a ping message. In this case, it decrements its reference count appropriately.

An immediate concern raised by this method is that of flooding the network with pinging messages. If every client had to ping a component periodically for every interface it held, large systems with just a few thousand components and clients would quickly saturate their network to maintain this protocol. To alleviate the burden on the network, DCOM uses a per-machine keep alive message. Regardless of the number of clients on a machine, a single ping message from that machine keeps all the connections alive. Further optimization is accomplished by a method called delta pinging. Instead of sending 100 client identifiers, meta-identifiers that represent all 100 are created. If the set of references changes, on the delta between the two reference sets is

```
COSERVERINFO ServerInfo ;

memset (& ServerInfo, 0, sizeof(ServerInfo));
ServerInfo.pwszName = L''MyRemoteServer'' ;

MULTI_QI mqi[3];
mqi[0].pIID = IID_IX;
mqi[0].pItf = NULL;
mqi[0].hr   = S_OK;
mqi[1].pIID = IID_IY;
mqi[1].pItf = NULL;
mqi[1].hr   = S_OK;
mqi[3].pIID = IID_IZ;
mqi[3].pItf = NULL;
mqi[3].hr   = S_OK;

HRESULT __stdcall CoCreateInstanceEx(CLSID_MyComponent,
     NULL,
     CLSCTX_REMOTE_SERVER,
     &ServerInfo,
     3,
     &mqi);
```

Figure 7.2: An example of CoCreateInstanceEx

sent. These ping messages can also be piggybacked onto regular messages. Lastly, the pinging protocol can be disabled when it is not required.

7.3.2 Remote Creation

DCOM also enables clients to specifically create a component on a remote server, using the *CoCreateInstanceEx* call, instead of CoCreateInstance which was discussed in section 6.3. Usage of this call is shown in figure 7.2. The COSERVERINFO structure contains the name of the remote server. Another important feature of DCOM is the MULTI_QI structure, which is passed in as the last argument to CoCreateInstanceEx.

Querying for interfaces on a local machine has relatively little overhead, with the use of DLLs and LPCs between processes. However, the network again creates additional constraints, since each call across a network can potentially introduce considerable delay. If clients had to repeatedly query for an interface across the network, it is conceivable that delay times would be inordinately

great, rendering client applications useless. The MULTI_QI structure tackles this situation by allowing the client to query for multiple interfaces with one call. The component returns the appropriate response by storing both the structure and the interface pointer of each request in the structure shown in the example. This method can be used either at creation time, as in figure 7.2, or through the *IMultiQI* interface, which is automatically implemented by the remote stub of the component.

7.3.3 The use of HRESULTS

A distributed system needs to handle error conditions in a graceful manner, in order to maintain a robust architecture. DCOM does this by requiring all methods to return a 32-bit error code called an *HRESULT*. There are certain tools in the Windows environment, and language conventions that allow HRESULTs to be converted into exceptions natural to the language, which can be dealt with in the standard way. These tools can also be used to "throw" exceptions, instead of returning a HRESULT in the standard way. DCOM also allows informative strings to be passed, containing descriptive explanations of the error. Many HRESULT codes are specific to the Windows operating system, and provide detailed error information. The MIDL compiler requires that all methods return a HRESULT. It is also possible to define one's own HRESULT codes.

7.4 Availability of DCOM

Perhaps the greatest advantage in favor of COM and DCOM is the wide base of users it inherits by virtue of its native environment - the Windows operating system. In fact, a large portion of the Windows and Windows NT operating systems themselves are written using COM components. According to Microsoft, COM is in use on over 150 million systems worldwide. [weba] DCOM naturally benefits greatly by the path forged by its predecessor, and is primarily targeted for Windows-based systems. DCOM is shipped as a part of Windows NT 4.0, and at the time of this study, a beta version is available for use with Windows 95. There are also efforts made by

other organizations, such as Software AG [2], to port DCOM to other platforms, including SUN Solaris, Linux and Digital Unix.

[2] http://www.sagus.com

Chapter 8

DCOM Implementation

To be able to compare and contrast the architecture and utility of DCOM with the other systems presented in this study, it is necessary to examine the practical aspects of creating a component and a client using this system. Although containing very simplistic examples, this will determine what is involved in creating an application using DCOM, and what practical considerations developers might face in using this system. The example is the same as was presented in chapter 5. The component we create will represent a Person object and will support one interface to perform manipulations on its age, called IAge. The IAge interface contains two member functions - SetAge and GetAge. This is pictured using the standard COM component diagram in figure 8.1. As we have seen, the person component may implement a number of interfaces. We will determine how to implement this component and its client in three ways:

1. **In Process:** The component is in the same process as the client.

2. **Local Process:** The component is in a different process on the same machine as the client.

3. **Remote Process:** The component and client are on different machines.

the language used for each implementation was C++, which has many inherent advantages for implementing COM components.

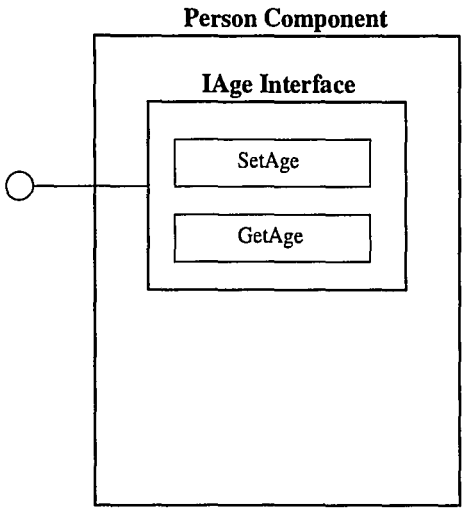

Figure 8.1: The Person Component Diagram

8.1 Defining the Interface

It was established in section 6.1.2 that an interface is simply defines a memory layout for a group of virtual functions in a vtbl. In C++, pure abstract base classes are classes that contain only pure virtual functions, marked with *=0*. These functions are not implemented in the base class, but in other classes that inherit it. Pure abstract base classes produce the exact memory layout defined by a vtbl, which makes it an ideal way to implement interfaces in C++. Figure 8.2 shows how the IAge interface is implemented. As shown, this interface fits the requirement of inheriting from IUnknown, as all interfaces must. The *__stdcall* prefix before function calls is a Microsoft-specific extension that tells its compiler how to use the stack for function calls, a detail which is not relevant to our purpose [Rog97]. The important features of this interface is that its methods are virtual, which allows different components to use this interface, and implement the methods as needed. This interface will be used by each of our implementation cases.

8.2 In Process

The first case we consider will show the client as an executable program with its own address space. The component will also be in the same address space, and be implemented in a DLL, as

An interface as an abstract base class

```
interface IAge : IUnknown
{
  virtual void __stdcall SetAge(int num) = 0;
  virtual int  __stdcall GetAge() = 0;

};
```

Figure 8.2: The IAge Interface

discussed in section 6.3.1

8.2.1 Implementing the Component

The first step is to implement the component. As shown in figure 8.4, this is accomplished in C++ by using a class that inherits from the abstract base classes of the interfaces it wishes to implement. In this case, the CPerson component supports the IAge interface. COM's naming convention is to prefix components with the letter 'C', and interfaces with the letter 'I', to clarify code and improve organization.

The methods of the component

The first three methods are those of interface IUnknown, discussed in section 6.2.1, which each component must implement. By virtue of C++ inheritance, these methods can be implemented in the CPerson class, which provides the implementation for the IAge interface as well. The inheritance is pictured in figure 8.3. The two methods of IAge are also implemented in CPerson, which sets the vtbl pointers to these functions. The data members are hidden from any client, and consist of a long int used for reference counting, discussed in section 6.2.2, and another integer to store our Person component's age.

The Addref and Release functions are simple enough, in that they simply increment and decrement the reference count. The QueryInterface method, discussed in section 6.2.1 is shown in figure 8.5. If a client queries for the IAge or IUnknown interface, a pointer to the IAge interface

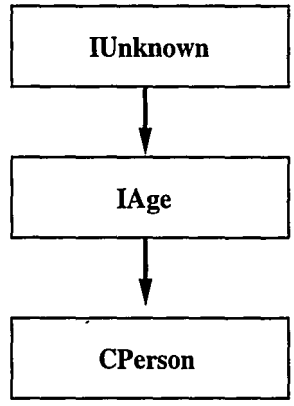

Figure 8.3: Cperson inherits the IAge interface, which inherits from the IUnknown interface.

```
class CPerson : public IAge
{
public:
  virtual HRESULT __stdcall QueryInterface (const IID & iid, void ** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

  //Interface Iage
  virtual void __stdcall SetAge(int num)
  {
    cout << "\nIn IAge interface, SetAge method.";
    age = num;
  }

  virtual int __stdcall GetAge()
  {
    cout << "\nIn IAge interface, GetAge method.";
    return age;
  }

  //Constructor
  CPerson();

  //Destructor
  ~CPerson();

private:
  long m_cRef; //Reference count
  int age;     //Data member for age;
};
```

Figure 8.4: An example of an interface and a class in C++

```
HRESULT __stdcall CPerson::QueryInterface(const IID & iid, void ** ppv)
{

  if ( (iid == IID_IUnknown) || (iid == IID_IAge) )
    {
      *ppv = static_cast<IAge*>(this);
    }
  else
    {
      *ppv = NULL;
      return E_NOINTERFACE ;
    }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK ;
}
```

Figure 8.5: The QueryInterface method for Person component

is returned. Since no other interfaces are supported by this client, all other queries set the return pointer to null and return an error message.

The Class Factory

We studied in section 6.3.3 that COM uses a class factory to create all the instances of a type of component. Our CPerson component's class factory is shown in figure 8.6. It is seen that the class factory itself is implemented as a component, and hence supports the IUnknown interface, as indicated by the first three functions it implements. It also supports the IClassFactory interface, which contains the methods to instantiate the component, and provide a server lock for the client. The implementations of these methods are in the entire program listing in appendix C.1 Once the component is compiled and linked as a DLL, Windows requires it to be registered using the *REGSVR32.EXE* tool in Windows NT 4.0. This exports the DLL entry points listed below, allowing the COM library to start the DLL, create the class factory, and shut down the DLL when it is no longer required:

- DllGetClassObject
- DllCanUnloadNow

```
class CFactory : public IClassFactory
{
public:
  virtual HRESULT __stdcall QueryInterface(const IID &iid, void ** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

  //Interface IClassFactory
  virtual HRESULT __stdcall CreateInstance(IUnknown * pUnknownOuter,
   const IID & iid,
   void ** ppv);
  virtual HRESULT __stdcall LockServer (BOOL bLock);

  //Constructor
  CFactory() : m_cRef(1) {}
  //Destructor
  ~CFactory () { cout << "\nClass factory:\t\tDestroy self."; }

private:
  long m_cRef;
};
```

Figure 8.6: The QueryInterface method for Person component

- DllRegisterServer
- DllUnregisterServer

8.2.2 Implementing the Client

Now that the component is ready, the client needs to be implemented. The entire code for the client program is in appendix C.2. The role of the client is relatively simple. It first initializes the COM library using the *CoInitialize* call, then creates the component using CoCreateInstance, as described in section 6.3. This method gives the client a pointer to the IAge interface of the CPerson component, with which it sets the age, and then retrieves the age to check the value. Once the client has a pointer to a component's interface, the interaction is just like regular C++ method invocations. After the client is finished with this interface, it must release it, which causes the component to decrement its reference count. The client and component are no longer connected. It is possible to cache interface pointers and use them later, however components

are free to move, or they may be deactivated. This is why COM clients are allowed the explicit ability to create a component when it needs its services.

8.3 Local Process

In the previous section, we implemented a client and an in-process component that was served from a DLL. The DLL contained entry points for the COM library, so that its class factory could be created, and it could in turn produce an instance of the component in the client's address space. When the client and component are in different processes, they each have their own address space, and are each implemented as separate executable programs. In this case, the COM library needs a way to create the class factory of the component.

To allow this to happen, specific entries are made in the Windows Registry , which basically map a class identifier to an executable program. With this in place, a client that needs to create a component uses *CoCreateInstance* as usual, and the COM library finds the executable program that serves this component, and inititates it. Once the server program is initiated, all the class factories it supports are created. The server then register these class factories with the COM library using the *CoRegisterClassObject* method, as shown in figure 8.7. The COM Library then invokes the CreateInstance method of the desired class factory, which provides an interface pointer to the newly created component. This pointer is then passed back to the client, as in the in-process case. After this, the client and component may communicate as if they are in the same address space. The COM library also maintains a private table of active class factories that have been created. If a client needs to create one of these types of components, the class factory is ready to use, and the server program does not need to be started.

Since the client performs the same call, and receives the same response as in the in-process scenario, this entire procedure is completely transparent to it. In addition, there is no change to the actual components. Only the configuration of the server program and the entries in the Registry need to be changed to switch from the in-process server to a local process server. As we shall see, the same thing holds true for the remote case.

```
BOOL StartFactories()
{

  IClassFactory* pIFactory = new CFactory ;

  if (pIFactory == NULL)
    {
      return FALSE;
    }

  HRESULT hr = CoRegisterClassObject(CLSID_PersonComponent,
           reinterpret_cast<IUnknown*>(pIFactory),
     CLSCTX_SERVER,
     REGCLS_MULTIPLEUSE,
     &dwRegister);
  if (FAILED(hr))
    {
      pIFactory->Release() ;
      return FALSE;
    }

  pIFactory->Release();
  return TRUE;
}
```

Figure 8.7: The CoRegisterClassObject method registers a class factory with the COM library

8.4 Remote Process

The separation of the client and component onto different processes in the previous section was the major step towards achieving a distributed system. Once this is accomplished, DCOM allows the two executable programs to operate over a network without any change to either one. To do this, the following steps are performed:

1. The client executable, server executable and the proxy DLL are copied to the remote system.
2. The server registers its class factories on the remote machine
3. The proxy DLL is registered on both machines using REGSVR32.
4. The DCOMCNFG.EXE utility is used to determine which machine the server will run on.
5. The server is executed on the remote machine.
6. The client is executed on the local machine.

By following these steps, we have simply changed the location of the executable program in the Registry, using the DCOMCNFG utility in Windows. The entry for the Person CLSID now contains the information to invoke a remote server whenever a client makes a request for this component. Although very simple, this method seems quite inefficient, if every remote server had to be explicitly dealt with in this manner.

DCOM also contains some extra functions, that allow the client to connect to a remote component. This is done by using CoCreateInstanceEx, as discussed in section 7.3.2. This call provides the client with a pointer to the desired interface of a particular component, with which the client and component communicate as if they are in the same address space. All the underlying details of connecting to a remote component, establishing an RPC connection, and maintaining the pinging protocol are handled by DCOM, and hidden from the developer.

Part IV

Comparison and Summary

Chapter 9

Qualitative Summary

We have covered three different approaches to constructing a distributed object system, each of which shed some light on our fundamental purpose of determining what the requisites of such a system are. MPI outlined a rudimentary approach to distributing objects over multiple processes, while CORBA and DCOM presented comprehensive architectures that are widely deployed in the market today. This section will present the understanding of distributed object systems that was gained as a result of this study. We will discuss the differences and similarities in the methodologies adopted by the systems studied, providing a qualitative comparison. The next chapter presents the quantitative analysis and the conclusions it reveals.

There are many angles from which these systems may be considered. First, we shall discuss the different issues that arise in a distributed object system, with a comparison of how the MPI-based system, CORBA and DCOM each handle them. Subsequently, we will draw conclusions about how each of these systems meet the general criteria we presented in the first chapter, of what is desired in a distributed system. In addition, we shall also compare the distributed object approach with traditional monolithic systems and socket programming, which have widely prevailed until the advent of this new technology, to stress the value of this new approach to software development.

9.1 Design Issues

9.1.1 What is the fundamental building block?

In any system, it is imperative to determine what is the fundamental unit of operation. In a distributed object system, this would instinctively be the object itself. However, different systems have different ways of defining an object and its role.

MPI

The sample system we developed is built upon the C++ definition of an object, since this was the language of implementation. Objects have data members and member functions, which present an interface to the client.

CORBA

A CORBA object is very similar to a C++ object, in that it contains data members and a set of member functions that determine its behavior. However, the implementation can be in any language for which a language mapping is defined. An class's member functions comprise its interface, through which clients may access and perform operations on data of particular object instances. CORBA objects support multiple inheritance and can be used polymorphically.

DCOM

In DCOM, the fundamental unit is the component, which is defined in 6.1.1. DCOM components can support a number of interfaces, each presenting a different view to the clients. DCOM interfaces consist of one or more functions. COM/DCOM specifies a binary standard for how interfaces must be laid out in memory, leaving the implementation to be in any language.

9.1.2 How are the services of an object described?

In an object oriented system, objects provide services which are used by clients. There needs to be a way for clients to determine what services are provided by each object.

MPI

Since we strictly used the C++ language for our implementation, the services of an object are determined by its member functions. Clients have access to the class declarations, and hence know which member functions the class supports. This method is quite inflexible, since the clients need access to the C++ class declarations.

CORBA

A contract between the client and the object implementation is defined in the IDL program. Both the client stub and the server skeleton are generated using this IDL description of the interface. This represents a promise of service to the client, and an obligation to be fulfilled by the object implementation.

DCOM

Generally, the interfaces are defined in a separate file, which is linked with the client program. In our example, which used C++, these were abstract base classes. These interfaces also represent a promise of service to the client and an obligation to the component. In addition, a client can query a component for a particular interface. If it is supported, the client will receive a pointer to the interface, which can be used to invoke its methods.

9.1.3 How do clients use new objects?

In an extensible system, it must be possible to add new objects, and allow them to be used immediately by clients in the system. This requires a method for clients to discover these new objects, find out what services they offer, and start using these services.

MPI

New objects are added by explicitly writing their class implementations, and recompiling the program. This object can be made to register with the registry tool, intimating it of its static instantiation function. Although this does not require any additional modifications to the registry object, the program needs to be recompiled, which is a drawback.

CORBA

A new object can be built and introduced into the system by having it register its interface in the IR. By using the IR, the client can discover new objects and their interface definitions. This provides the information about the services the new object offers. The client can then use the DII to construct and dispatch an invocations to the new object.

DCOM

DCOM allows clients to query components for a particular interface. If a new component supports this interface, the client can use it immediately. However, the client can only query for interfaces that it knows of at compile time. DCOM solves this by allowing the client to use the type library generated by IDL programs to dynamically query for interfaces which it does not know of at compile time. This is similar to the DII in CORBA.

9.1.4 Process Boundaries

We have seen that a distributed object system needs to cross some formidable barriers that traditional systems can ignore. First, there is the inter-process leap, which is necessary to enable two processes to communicate and allow objects in these processes to interact. Next comes the inter-machine boundary, which expands this interaction over a network. After studying these three systems, it is clear that there are only two ways to deal with inter-process and inter-machine boundaries:

- Bring the target object into the client's address space **OR**

- Create the illusion that they exist in the same address space

MPI

The system developed in part I used a variation of the first approach, by creating a replica of the target object in the client's address space. This has problems with maintaining unique object identity and accurately reflecting changes in the original object.

CORBA and DCOM

Both these systems use the second approach, since this provides practically results in a usable and efficient distributed object architecture.

9.1.5 Can the client and object be developed separately?

In a dynamically evolving system, separation of the client and object implementation allows each one to evolve at their own rate, and gives the system the flexibility to change without being taken down and completely rebuilt. This is done using interfaces between client and objects, as we have seen. To facilitate this, a distributed object system must allow the independent development of client and objects. This introduces a compromise in runtime performance. A statically linked program is faster than one that has to communicate with another process, or load in modules from a library.

MPI

Since we used the SPMD paradigm for our sample example, this is not possible. If this were to be a truly usable system, it could be implemented in MPMD style, which is supported by MPI, and the objects and clients could be split into different processes, that would each run as a separate executable program.

CORBA

This is one of the reasons that fueled the development of architectures such as CORBA and DCOM. The client and object programs can be coded separately, and be executed as separate executables. The IDL definitions allow them to fit together at run-time. Both MICO and OmniBroker implement their ORBs as libraries which are loaded into these executables at runtime. Most vendors adopt this approach. MICO even allows the object implementation to be loaded as a module into the client process, which yields better run-time performance. However, this feature is not mentioned the CORBA specification. Instead, there are different modes in which the BOA can activate objects, as discussed in section 4.3.4.

DCOM

Built upon the component technology of COM, DCOM allows components to be separate units which can be developed independently, fitting together through their interfaces. COM/DCOM allows components to be in four modes, as discussed in section 6.3. Of these, an in-process server is as fast as a statically linked application. A local process server and remote server are implemented as executable programs, and compromise runtime speed for flexibility in application evolution.

9.1.6 How is inheritance achieved?

Since we are modeling a distributed object system, it needs to incorporate support for object oriented principles. However, these systems adopt different approaches to these principles, as presented in the way inheritance is handled.

MPI

Normal C++ inheritance can be used. Children inherit both implementation and the interface from parent classes.

CORBA

CORBA uses the C++ style of inheritance. Children inherit both the interface and the implementation of parent classes. CORBA objects can also multiply inherit from many parents. This is reflected in the IDL definitions as well, in which interfaces can multiply inherit from many interfaces. All CORBA objects are part of a class hierarchy, since each object implementation inherits from CORBA::Object, through its skeleton class and its abstract base class in that order.

DCOM

The C++ style of inheritance is not used in DCOM. Only interface inheritance is explicitly supported. In addition, multiple inheritance is not supported at the interface level. Instead, this is achieved by allowing components to support multiple interfaces. DCOM interfaces are also part of a interface hierarchy, since all interfaces inherit from IUnknown. DCOM is basically oblivious to the relationships between component classes [CYY]. Implementation inheritance is not supported, but is simulated using containment and aggregation, as covered in section 6.4.

9.2 Implementation Issues

9.2.1 How are objects identified?

One of our requisites for a distributed object system was that it support any number of objects. In this scenario, there needs to be some way to identify the objects, for a client to use its services. In addition, each instance of an object needs to be uniquely identified if a client's operation depends upon the state of the object.

MPI

In our example system, objects were identified by a simple string that uniquely described the object. We used human readable names, which work fine for small numbers of objects and a

small scale of distribution. However, it becomes very difficult to maintain their uniqueness as the system grows.

CORBA

CORBA assigns an object reference to every object at the time of its creation, which remains valid until the object is deleted. Clients use these object references to access the methods on an object instance. This allows a client to recover the exact instance it desires.

DCOM

Components are given a unique CLSID, which describes the type of component. Individual instances of a component are never accessed directly by a client in DCOM. The only view presented to the client is through the component's interfaces. Interfaces are also given unique IIDs, and a client is given a pointer to an interface to access the component. This pointer is a client's only way to access a particular instance of a component.

9.2.2 How do clients get the object identifier?

To access an object, a client needs use this identifier. But how does it get the identifier itself?

MPI

The string identifier of the objects is sent with its DEX description. When the client process gets this description, it extracts the object identifier, and instantiates the proper object in its own address space. Our example only focused on two processes, but there could conceivably be multiple processes which each serve different objects. Using the collective communication feature, a client could then send a broadcast message requesting for a particular object. The process serving this object could then stream it into the client's process in the manner described in chapter 3.

CORBA

The object identifiers are stringified and made available to the client either through files or some other method. The client converts this string to an object reference, using a standard CORBA function. Using this reference, the client can access the object.

DCOM

The client specifies a CSLID when creating a component. This CLSID can be statically linked into the client's program. Alternately, the CLSIDs are stored in the Registry as strings. DCOM supports functions for searching the Registry, and converting a CLSID to and from a string.

9.2.3 How are objects created?

In a dynamic distributed system, there must be a flexible and generic run-time mechanism of creating objects. This allows the dynamic addition of new objects, as an when needed.

MPI

Each class in this system has a static member function that instantiates the class and returns an object of that type. This member function is called whenever an object is needed.

CORBA

CORBA objects are instantiated by its server program when it is activated by the BOA. A client normally tries to invoke a method on an object using an object reference, which triggers the ORB to send a message to the BOA asking it to activate the server and the object implementation.

DCOM

DCOM components are created by a class factory that is implemented by the component programmer. A client can either make a CoCreateInstance or CoCreateInstance call, or create the

class factory directly. This factory creates an instance of a class, queries it for the interface desired by the client who created, or triggered the creation of the class factory, and returns an pointer to this interface. Similar to our MPI-based system, component developers can implement a CreateInstance function to instantiate the components. The class factory resides in a server program or a DLL.

9.2.4 How are servers registered?

Since there may be a large number of objects located in different server programs, even spread over different machines, there needs to be a generic way to locate a particular server. This is done by maintaining information about the machine and path of the server program.

MPI

In our example system, there were only two participating processes, one acting as the client and the other as the server of the objects. In this case, there was no need for such a service.

CORBA

Server programs are registered in the Implementation Repository, which associate an interface name with the path of the server executable. The ORB and the BOA use this to find and activate a server program to fulfill a client request.

DCOM

The Registry performs this function in DCOM. It associates the CLSID of each component with the path of the executable server program or the DLL which serves the component. This can also specify a machine name, in case of a remote server.

9.2.5 Connecting client and object

In both of the approaches detailed above, an actual connection is made between the client and the object, which allows them to communicate. The first method does this by using a common address space. The second method creates the illusion by establishing an actual network connection, at the lowest level.

MPI

The connection is made via a common address space.

CORBA

CORBA does this by creating a socket endpoint at the object implementation side when the object is instantiated and an object reference is generated for it. This object reference contains information about the machine name, a TCP/IP port number, and an object key, and is passed through the ORB to the client side. At the client side, the stub extracts this information and creates a socket connection, which is used in all further communication with the object implementation. [CYY]

DCOM

A DCOM server that creates an object, an object reference (OBJREF) is created to represent the interface pointer, and contains information about the implementation scope of the object, which could be a machine, a process, or a thread within a process. This is resolved to a single interface of an component on a particular server. This OBJREF is carried to the client side, and the client proxy uses this to establish an RPC channel to the component. This RPC channel is then used in all further communications between client and component. [CYY]

9.2.6 Passing Information

We saw that the fundamental problem is of having different address spaces, which makes it impossible for an object in one process to directly invoke the method of an object in another process. To bridge this gap, each system relies on a standard method of passing information between processes, which allows the marshaling and unmarshaling of parameters.

MPI

In the first part, the DEX standard took care of marshaling all data into a character stream, and unmarshalling it at the receiving end. The notable point is that the entire object is marshaled into this character description, not just parameters.

CORBA

CORBA defines a Common Data Representation (CDR) to marshal parameters [CYY]. This defines representations for all IDL data types, and is used by the GIOP to transfer data [spe96].

DCOM

DCOM uses the Network Data Representation (NDR) defined in OSF's DCE to marshal parameters of a function call [CYY]. It supports a standard technique for marshaling, and also allows components to implement an IMarshal interface, to specify what data is marshaled, and how it is done.

9.2.7 How are errors handled?

A good distributed system needs to be robust, especially considering the various points of possible failure, as seen in section 7.2. To achieve this, errors need to be reported and handled properly.

MPI

Although our system did not explicitly include code for reporting and handling errors, both C++ and MPI provide extensive support for this task. MPI defines several error codes that can be detected using its library functions, which would handle the transport layer errors in our system. In addition, C++ exceptions could be used for errors at the application level.

CORBA

CORBA provides extensive support for C++ type exceptions, which can be used in the IDL definitions of interfaces, and at the host programming language level. Exception handling is provided for in language mappings for host languages. CORBA defines some exceptions specific to its operation, and allows the user to define their own exceptions as well.

DCOM

As seen in section 7.3.3, DCOM requires that all methods return a 32-bit error code, or a HRESULT. This can be used by certain tools that convert these codes into exceptions in a host language. These codes can also be used to generate strings describing error conditions.

9.3 General Conclusions

Evidently, there are many issues to consider in the design and implementation of a distributed object system. The study of these three systems presented several insights to these fundamental issues, as summarized above. Having seen how these systems attack these situations, we shall now assess their performance in lieu of the general qualities that are desired in a distributed object system, as presented in the first chapter.

9.3.1 Ease of Development

In the past, a typical object oriented application that had to communicate between processes would have to maintain the objects, define their interaction, and also enable them to communicate over the network. Practical constraints of time and resources shackled this approach by requiring developers to code for a particular type of transport, such as TCP sockets. Once this is built into the objects of the system, any change is impossible, or at least too expensive to consider. There are two major ideological shifts introduced by the distributed object systems we have presented. First, the monolithic application is broken into separate components that can evolve at their own pace, lending to a robust and simultaneously dynamic entity. Secondly, these systems depart from the traditional approach in which developers were straddled with the responsibility for implementing the application logic, the distributed protocols, and the lower level transport details. These distributed systems abstract away the lower two layers, allowing developers to focus solely on their immediate purpose - the application logic. By doing this, a degree of flexibility is also introduced into the system, to allow for future changes in lower layer technology. In this sense, these systems greatly simplify the overall development process, and show their utility in increasing measure over time. However, each of the systems covered in this study have a fairly steep learning curve, especially for those trained in the traditional and monolithic approaches to developing applications.

MPI

To develop the MPI system presented in Part 1, first the capabilities of MPI itself have to be understood, including its various modes of operation and interprocess capabilities. Following this, the concept of describing an object using DEX is needed to implement any object in our system. In a truly object oriented system, an object manager would be needed to describe and instantiate the various objects. However, this is a one-time effort, and only needs to be maintained with the addition of objects.

CORBA

Once one is familiar with IDL and its features, the effort of developing a system using CORBA is greatly simplified. In addition, this standard strictly adheres to the principles of object oriented programming, and defines concrete language mappings for C++ and various other languages. Therefore, coding objects and their clients is largely intuitive for those familiar with object oriented principles. The role of the ORB, the repositories and the utility of dynamic invocations require adjustment from traditional programming. however, and only some hands-on experimentation can acquaint one with their intricacies. Since the OMG is such a large and widespread consortium, CORBA is the collective effort of many minds from diverse disciplines, forcing the concepts to be less esoteric and generally understandable.

DCOM

This approach to the distributed object architecture contains many efficient optimizations for its native environment - the Windows operating system, and hence makes it slightly more difficult to grasp for those not familiar with it. The basic concepts are very simple and plain, such as the memory layout of a vtbl. However, DCOM defines a binary standard, and thus provides no language mapping like CORBA. The result is that the developer is free to do what he or she wishes, as long as the binary specifications are met. While this allows flexibility for the experienced developer, new entrants can be found floundering for some foothold! There are some departures from traditional object oriented concepts. For instance, the interface is the central figure in this architecture, not the object itself. However, once the strict coupling of object oriented principles and the C++ language is abandoned, COM's techniques become relatively easy to grasp. The actual code required to create our DCOM example is considerably more than that of the same example in CORBA.

9.3.2 Compatibility

By their very nature, distributed applications need to span a variety of environments, especially with the increase in heterogeneous collections of workstations, and the growing presence of the

Internet. A sound distributed system should be compatible across different operating systems, as well as underlying hardware architectures and network protocols. Again, contrasting this with the traditional style of programming, an application written for a particular operating system is generally difficult to port to another operating system. Hardware differences are more difficult to account for, and changes in network protocols could result in the entire reworking of an application. This is why porting large systems is often a herculean task involving many minds and extended durations. While any system can eventually be changed to operate on a different environment, a broadly compatible system eases this transition greatly. Here's how the three systems we covered stand up to this.

MPI

Since MPI is actually a specification, the concepts can theoretically be implemented in any environment. Practically, the implementations that we used, MPICH and OOMPI, is supported on many operating systems, including Linux, Sun Solaris, and Digital Unix. The sample application presented in chapter 3 was compiled and run under an Intel Pentium running Linux, and worked in the same manner without any changes on a Sun Solaris machine. MPICH also allows its programs to be run on a heterogeneous collection of workstations. The DEX standard simply describes an object in a character stream, and can be implemented on any platform. Inter-language compatibility in our system is restricted by MPI, which only supports FORTRAN 77, C and C++.

CORBA

Again, CORBA being a specification, it is largely up to the implementors to determine how easily compatible their product is. Compatibility between the ORBs of different vendors is a major complaint against CORBA. It was seen in chapter 5 that server programs were quite different between ORBs. While the IDL itself is fully compatible across ORBs, the developer's code for the client and the object implementation inevitably has to be changed to suit the particular conventions of each vendor's ORB. In large systems, this could involve major reworking. A point to note is that it would generally be uncommon to shift your application from one ORB

to another, so inter-ORB portability is not as important an issue as inter-ORB compatibility. The latter is provided for by the IIOP standard. CORBA is supported on a wide number of operating systems, including various flavors of Unix, as well as Windows 95 and Windows NT. In the sample presented in chapter 5, the client and object both worked on different platforms. A client on a Linux machine could access an object on a Solaris machine, and vice versa. CORBA applications can be developed in any language that has a mapping defined, currently including C, C++, Java, Smalltalk and Python.

DCOM

Since DCOM relies upon its large base of Windows systems, and COM initially was a internal effort to simplify development, compatibility of this standard across platforms was not an initial consideration for its architects at Microsoft. There are efforts undertaken by other companies, such as Software AG, to port DCOM to other platforms, but even this is done by completely recreating the Windows environment, including the Windows registry, to allow interoperability with Windows systems. These ports allow interoperability between platforms, but leave out some features on non-Windows systems. For instance, a COM client on a Windows machine can communicate with a COM component on a SUN Solaris machine, but no GUI support is available as yet for these components under Solaris. DCOM is the most flexible for inter-language compatibility since it defines a binary standard, and the source-code can be in any language that allows the creation of vtbls and invocation of functions through pointers.

9.3.3 Transparency

A good distributed system presents the entire collection of machines as one single system to the developer and the objects in the system. A client should not have to take special action in order to access a remote object. Knowledge of location can promote dependency upon these locations being fixed, and hence makes a system less robust. If the clients and objects in a distributed system operate without any knowledge of location, then they can be moved around without any effect on their operation. As is seen in this study, an object oriented approach inherently allows for the encapsulation of details such as location. The client's only view of an object is presented

by the methods it supports, while the object is only concerned with implementing these methods. This can only happen if the lower level details of creation, connection with the client, etc. are delegated to some external authority.

MPI

The mini-system we modeled can be considered to transparent with respect to location, since the client and server processes are both oblivious of the machine they are on, or even whether they are in the same machine. The only important factor in this respect is that one process is the client, and the other is the server, which is determined by their rank. The location of their execution is determined by a MPI configuration detail, independent from the application logic. This was one of the reasons for using MPI to hide the lower level details of transport over the network and process execution.

CORBA

Location transparency is achieved in this system by virtue of the client routing all requests to the ORB, which then finds the object implementation and delivers the request to it. Object implementations also return responses to the ORB, which ferries it to the client. Although a socket connection is established between the client and object implementation, this is done below the layer of the client and object programs, in the stub and skeleton respectively. Because of the ORB acting as an intermediary, location is hidden from the developer, as well as the clients and objects.

DCOM

DCOM provides location transparency by having the client always send its requests to an object in its own address space. This might be either a proxy from a DLL, or the component itself, if it is served from a DLL. In other words, the client does nothing different when the component is in its own process, or if it is on another machine. The actual implementation of the object methods also are free from location details. It merely has to fulfill the obligation stipulated by

the component's interfaces. In this system, the external authority which handles the layers below the application logic consists of the COM and DCOM libraries, along with the Windows Registry.

9.3.4 Reliability

To be practically useful over a range of large applications and long durations, it is vital that a system prove its reliability. A distributed object system should be more reliable than a single-machine system, which has a single point of failure. If an object server crashes, the system must be able to recover, and still service client requests. To a large extent, this type of reliability needs to be incorporated into the applications themselves. However, a good distributed object architecture can lend to this goal. Tanenbaum defines three tenants of reliability [Tan95]:

- Availability: The fraction of time that the system is usable. This can be improved by replicating critical components.
- Security: Resources must be protected from unauthorized usage.
- Fault tolerance: Failures can be effectively masked and recovered from.

MPI

This system is not designed to meet the scale of providing reliable services, but to understand the choices and compromises in constructing a distributed object system. However, in this effort, it was seen that in order to make this a reliable system, it would be necessary to introduce an element of redundancy, by having more than one registry object, for instance. Security is not addressed at all, except for the fact that only the member functions of an object have access to its data members. This system is also not fault tolerant, considering its tight coupling of the two processes involved. A larger system could be based on the same design, in which one process could mask the failure of another.

CORBA

Although CORBA presents a comprehensive distributed object architecture, it does not fulfill all the three aspects of reliability. If the ORB itself crashes, the entire system becomes a disjoint set of entities. Depending upon the ORB as a sole central authority makes it difficult to mask failures within the ORB and make the system truly fault tolerant. However, the exception handling provided by CORBA and its ability to initiate server programs that are inactive provide for a considerably robust system. There is extensive support for security, outlined in CORBAservices, which is not covered in this study.

DCOM

DCOM achieves a fair degree of fault tolerance by giving the component the ability to delete itself when its reference count drops to zero. This coupled with the pinging protocol separate objects from failures that can occur in the network or the client side. This system also achieves a variation of redundancy, since each machine has a separate Registry, which can control its local clients and components. One registry failure should not affect other members of the system. DCOM also has extensive support for security, using the security framework of Windows NT, and several other methods, which are not covered in this study.

9.3.5 Scalability and Flexibility

Any system must have the ability to be extended over time. Distributed object systems rely on the ability to be able to add an unspecified number of objects into the system, while maintaining its performance. Another factor related to this is flexibility, which can be defined as the ability to change the size of the individual modules of an application, and the way they interact. In large systems, requirements and conditions are dynamically changing. For instance, as a banking operation grows, it may be necessary to split its deposits module into several modules, for foreign deposits, domestic deposits, corporate deposits, etc. Similarly, it may be necessary to aggregate modules into a larger function. Traditional applications normally undergo major changes as re-

quirements increase. The component technology introduced by DCOM and CORBA will greatly alleviate this problems, allowing applications to evolve naturally in a logical manner.

MPI

Our rudimentary system works fine for a predefined problem space. While it is possible to add new objects into the system, and have them replicate themselves in a different process in the same manner, the limitations of this approach become glaringly apparent when large numbers of objects need to be added. The registry object presents a bottleneck which would slow the entire system down. Another problem is that of duplicate identity, which we ignored in order to expose other points. To make this system truly flexible, and to allow objects the freedom to dynamically change the way they interact with each other requires a lot of work.

CORBA

Scalability and flexibility are the main reasons why entire architectures were developed for distributed objects, instead of adopting basic methods as displayed in our MPI system. CORBA was developed to support large systems with unspecified numbers of objects and complex relationships between different modules.

Chapter 10

Quantitative Analysis

The three systems presented in this study each adopt different methods to allow the integration of distributed objects into a single system. By examining the intricate features of each approach, we are able to determine what the requisites of a distributed object system are, and assess how each of these systems meet our criteria, as detailed in the previous chapter. In order to ascribe concrete measures for evaluating the efficacy of these systems, it is necessary to get a feel for how they actually perform in their task. To compare the performance of the MPI-based system, CORBA and DCOM, we designed a series of common tests applicable to all three systems.

The controlled environment established for the tests consisted of two Intel Pentium Pro 200 MHz machines that ran the Linux 2.0.32 operating system for the MPI-based and CORBA systems, and Windows NT 4.0 for DCOM. These machines were isolated from network traffic, and connected using a 10 Megabit Ethernet as well as a 100 Megabit Ethernet connection. Both these network interfaces could be used exclusively, to ensure the accuracy of the tests.

The basic action in a distributed object system is of a client making an invocation on an object. The object might be remote, on the same machine, or in the same process, if allowed by the system. These tests all measure the time taken for a client to make invocations on an object, with response to the following factors:

- **Various sizes of objects**

- **Number of invocations**
- **Utilization of network resources**

10.1 Various sizes of objects

A distributed system needs to pass information between different processes in an efficient manner, to allow the various components to communicate. Each of the three systems in this study adopt a different method to accomplish this, as seen in section 9.2.6. In order to test the performance of these methods, a client makes an invocation on an object, and receives the different sizes of responses. This invocation is timed for the following sizes of responses sent by the object: (Results are shown in figure 10.1)

- single integer
- complex structure composed of sub-structures
- 1 KiloByte block of data
- 250 Kilobyte block of data
- 500 KiloByte block of data
- 750 Kilobyte block of data
- 1 MegaByte block of data
- 5 Megabyte block of data
- 10 MegaByte block of data

It is seen that each system has a different response to the increase in the sizes of responses that the object sends to the client. This reflects the overhead imposed by the system on object responses. This overhead can also be measured in a different way, as in the section below.

Size of response from object

System	int	struct	1 KB	250 KB	500 KB
MPI	.002378	.005634	.010453	.054782	.756523
CORBA	.006388	.008499	.010028	.234901	.559982
DCOM	.007123	.007214	.012981	.135168	.531986

System	750 KB	1 MB	5 MB	10 MB
MPI	.084567	1.009812	2.983413	4.348712
CORBA	.84567	2.344554	5.324351	8.123256
DCOM	.719865	3.412982	7.129876	10.218976

Figure 10.1: Time taken for invocations (in seconds)

Number of Invocations

System	single	100	1,000	10,000
MPI	.000345	.209875	2.813456	21.009824
CORBA	.008677	.324564	3.196562	32.010894
CORBA	.008712	.358123	4.129844	29.010894

Figure 10.2: Time taken for invocations (in seconds)

10.2 Number of Invocations

When a client makes an invocation on an object in a distributed system, there are many steps involved for the request to reach the object, and the response to be delivered to the client. Each system can be compared in its ability to optimize this interaction, and minimize the time needed for a method invocation. This can be measured by repeatedly making a number of invocations, and examining the system's response. This was done by making:

- a single invocation
- 100 continuous invocations
- 1,000 continuous invocations
- 10,000 continuous invocations

The results of this are seen in figure 10.2

10.3 Usage of Network

Since a distributed system abstracts the lower layers of network connection and transport from the clients and objects, it needs to handle these issues itself. Messages must be accurately and speedily presented to and retrieved from the underlying network layer. Depending on how this is done, each system can utilize the network with different degrees of efficiency. This was measured by repeating both tests mentioned in the previous two sections over a 100 Megabit Ethernet connection, so that we could compare the utilization of a :

- 10 Megabit Ethernet connection
- 100 Megabit Ethernet connection

The results for the 100 Megabit connection are seen in 10.3. It can be observed that no system shows considerable improvement when dealing with small amounts of data, and single invocations. This is because the network traffic is too sporadic and bursty to fully utilize a 10 Mb connection. The 100Mb connection will offer a speedup only if the application can exceed the capability of the 10Mb connection. This does not occur until the object sends larger sized responses to the client. Note that the CORBA tests show no considerable speedup between the 10Mb and 100Mb connections for a 250 KB packet, while the MPI-based system does shows a speedup. This indicates that CORBA's overhead prevents it from saturating a 10Mb connection. Only after the packet size increases to 1MB does CORBA exceed the capacity of the 10Mb connection.

Size of response from object

System	int	struct	1 KB	250 KB	500 KB
MPI	.002231	.005565	.012130	.051215	.071298
CORBA	.006120	.008123	.011012	.200912	.498123
DCOM	.007128	.007323	.049193	.310191	.498101

System	750 KB	1 MB	5 MB	10 MB
MPI	.08213418	1.008123	1.831232	2.998765
CORBA	.771287	1.548912	2.122420	4.123218
DCOM	.783219	1.98122	2.820910	5.009123

Number of Invocations

System	single	100	1,000	10,000
MPI	.000320	.209135	2.003876	8.91282
CORBA	.003912	.219013	2.001219	12.08049
CORBA	.007609	.298100	2.831342	15.400981

Figure 10.3: Varying Network Speeds

Bibliography

[art96a] The component object model: Technical report. *Dr. Dobbs Journal*, 1996.

[art96b] Dcom technical overview. 1996.

[CYY] P. Emerald Chung, Huang Yennun, and Shalini Yajnik. Dcom and corba side by side, step by step, and layer by layer.

[OHE96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide.* John Wiley and Sons Inc, 1996.

[RK] Steve Robinson and Alex Krassel. Components. Technical report, Panther Software.

[Rog97] Dale Rogerson. *Inside COM.* Microsoft Press, 1997.

[Sie96] Jon Siegel. *CORBA Fundamentals and Programming.* John Wiley and Sons Inc, 1996.

[SOHL+96] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* The MIT Press, 1996.

[spe] The data exchange standard. Technical report, The Image Understaning Environment.

[spe96] The common object request broker: Architecture and specification. Technical report, Object Management Group, 1996.

[Tan95] Andrew S. Tannenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[weba] Com: Delivering on the promises of component technology. Technical report, Microsoft.

[webb] http://www.omg.org/about/whorwe.htm. Technical report, Object Management Group.

Appendix A

MPI examples

A.1 Example using the Generic DEX API

```
1
2    #include <iostream.h>
3    #include <strstream.h>
4
5
6    #include "dexinc.h"
7    #include "oompi.h"  //Use Object Oriented MPI
8
9    #include "person.h"
10   #include "animal.h"
11   #include "vehicle.h"
12   #include "registry.h"
13
14   template <class T> void Writeobj ( T &obj ,
15                                      DEX_table & table,
16                                      DEX_class & cls,
```

```
17                                      DEX_object * objp,
18                                      int tb_count)
19  {
20    obj.fill_dextable_from_obj (obj, table, cls, objp, tb_count);
21  }
22
23  main (int argc, char * argv[] )
24  {
25    int i, tag=99;
26
27    char msg[2000];
28
29    OOMPI_COMM_WORLD.Init(argc, argv); // initialize oompi stuff
30
31    char processor_name[MPI_MAX_PROCESSOR_NAME];
32    int namelen;
33    MPI_Get_processor_name(processor_name, &namelen);
34
35    int rank = OOMPI_COMM_WORLD.Rank();
36    int size = OOMPI_COMM_WORLD.Size();
37
38    //this is what sets up the Ports in OOMPI. It determines which port
39    //you are receiving from, and which port to send to.
40    int to = (rank +1) % size;
41    int from = (size + rank -1) % size;
42
43    char request[10];
44
45    if (rank == 1) //Server Process. This will listen for requests
46      {
```

```
47          DEX_table table;
48
49          ostrstream * ofs = new ostrstream; // Use String buffer stream.
50          DEX_writer writer;
51
52          table.header.dex_version = "1.0";
53          table.header.producer_tag = (string)"Santosh";
54          table.header.producer_version = (string)"0.0";
55          table.header.comment = (string) "Just trying out";
56
57          int tb_count = 0;
58
59          DEX_class dexclass;
60          DEX_object * dexobj = new DEX_object;
61
62          OOMPI_COMM_WORLD[from].Recv(request, 10);
63          printf ("\nServer on %s received request for %s\n",
64                  processor_name, request);
65
66          if (!strcmp(request, "person")) {
67            person obj ("Santosh", "Sreenivasan", 22, 68);
68            //Writeobj will be a templated func
69            Writeobj (obj, table, dexclass, dexobj, tb_count);
70          }
71          if (!strcmp(request, "animal")) {
72            animal obj ("Human", 145, 22);
73            //Writeobj will be a templated func
74            Writeobj (obj, table, dexclass, dexobj, tb_count);
75
76          }
```

```
77          if (!strcmp(request, "vehicle")) {
78            vehicle obj ("GPS 19T", "Honda", "Accord");
79            //Writeobj will be a templated func
80            Writeobj (obj, table, dexclass, dexobj, tb_count);
81          }
82          writer.table(table);
83          writer.open (*ofs);
84          char st[3];
85          sprintf (st, "%d", tb_count);
86          writer.put (st);
87
88          ofs->freeze(0);
89
90          char * result = ofs->str(); // put dex output into a string
91          int size = strlen(result);
92          cout << "\n\n\nThe Result is: \n" << result;
93
94          strncpy (msg, result, size);
95          OOMPI_COMM_WORLD[to].Send(msg, size);
96          printf ("\nMessage sent from %s\n",processor_name);
97
98        }
99      else if (rank == 0) //Client process
100       {
101         char ch;
102         char req[10] = {'\0'};
103         printf ("\nClient Process on %s ", processor_name);
104         printf ("\nEnter (P), (A), or (V): ");
105         scanf ("%c",&ch);
106
```

```
107         switch (ch)
108           {
109           case ('A', 'a'):
110             {
111               strcpy (req, "animal");
112               break;
113             }
114           case ('P', 'p'):
115             {
116               strcpy (req, "person");
117               break;
118             }
119           case ('V', 'v'):
120             {
121               strcpy (req, "vehicle");
122             }
123           }
124         printf ("\nSending request for %s\n", req);
125
126         // ********** Timing begins here ******************
127
128         OOMPI_COMM_WORLD[to].Send(req, 10);
129
130         OOMPI_COMM_WORLD[from].Recv(msg, 2000);
131         printf ("\nObject %s received on %s\n",req ,processor_name);
132
133         registry reg;
134         void * (* func_ptr) ();
135
136         DEX_reader reader;
```

```
137         // Input string buffer stream
138         istrstream * ifs = new istrstream(msg);
139
140         reader.open (*ifs);
141
142         while (!reader.eof()) {
143           string readobj = reader.get();
144           DEX_table table = reader.table();
145           int pos = atoi(readobj);
146
147           string name = table.objects[pos]->class_name;
148           (void *)func_ptr = reg.find_inst_func(name);
149
150           cout << "\nName: " << name << endl;
151
152           if (!func_ptr)
153             cout <<"\nNo match!!";
154           else
155             {
156               printf ("\nClient on %s displaying object %s",
157                       processor_name, req);
158               if (name == "person")
159                 {
160                   person * newobj = (person *)(* func_ptr)();
161                   newobj->fill_obj_from_dextable (table, pos);
162                   newobj->disp();
163                 }
164               if (name == "animal")
165                 {
166                   animal * newobj = (animal *)(* func_ptr)();
```

```
167                 newobj->fill_obj_from_dextable (table, pos);
168                 newobj->disp();
169               }
170             if (name == "vehicle")
171               {
172                 vehicle * newobj = (vehicle *)(* func_ptr)();
173                 newobj->fill_obj_from_dextable (table, pos);
174                 newobj->disp();
175               }
176           }
177       }
178       // ************ Timing ends here ??? ***************
179     }
180   OOMPI_COMM_WORLD.Finalize();
181 }
182
```

Appendix B

CORBA examples

B.1 MICO implementation

B.1.1 Server program

```
1   #include <fstream.h>
2   #include <unistd.h>
3   #include "person.h"
4
5   class Person_impl : virtual public Person_skel {
6     CORBA::Short age;
7   public:
8
9     Person_impl()
10    {
11      age=0;
12    }
13
14    void SetAge (CORBA::Short num)
```

```
15      {
16        cout << "\nIn SetAge";
17        age += num;
18      };
19      CORBA::Short GetAge() {
20        cout << "\nIn GetAge";
21        return ++age;
22      }
23    };
24
25    int main( int argc, char *argv[] )
26    {
27      CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
28      CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");
29
30      Person_impl *per = new Person_impl;
31
32    boa->impl_is_ready (CORBA::ImplementationDef::_nil());
33        return 0;
34    }
35
```

B.1.2 Client program

```
1     #include "person.h"
2
3     int main (int argc, char *argv[] )
4     {
5
6       // ORB initialization
7       CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
```

```
8       CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");
9
10      assert (argc == 2);
11
12      CORBA::Object_var obj = orb->bind ("IDL:Person:1.0", argv[1]);
13      assert (!CORBA::is_nil (obj));
14
15      Person_var client = Person::_narrow(obj);
16
17      client->SetAge(5);
18
19      cout << "\nThe age has been set to: " << client->GetAge();
20
21      return 0;
22  }
```

B.2 OmniBroker Implementation

B.2.1 Server program

```
1   #include <OB/CORBA.h>
2   #include <OB/Util.h>
3   #include <person_impl.h>
4
5   #include <stdlib.h>
6
7   #ifdef HAVE_FSTREAM
8   #   include <fstream>
9   #else
10  #   include <fstream.h>
```

```
11  #endif
12
13  #define STR_SIZE 1048
14
15  main(int argc, char * argv[], char * [])
16  {
17
18    try
19      {
20
21        CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
22        CORBA_BOA_var boa = orb->BOA_init(argc, argv);
23
24        char buf[1048*1048]={'a'};
25
26        int n;
27        for (n=0;n<STR_SIZE;n++)
28          buf[n]='1';
29
30
31        buf[STR_SIZE]='\0';
32        cout << "\nIn Server.cpp. Buf size is: " << strlen(buf) << endl;
33
34        Person_var per = new Person_impl(0, buf);
35        CORBA_String_var str = orb->object_to_string(per);
36        const char * refFile = "person.ref";
37
38        ofstream out(refFile);
39        out << str << endl;
40
```

```
41          out.close();
42
43          boa->impl_is_ready(CORBA_ImplementationDef::_nil());
44        }
45    #ifdef __GNUG__
46      catch(CORBA_COMM_FAILURE& ex)
47    #else
48        catch(CORBA_SystemException& ex)
49    #endif
50          {
51            OBPrintException(ex);
52            return 1;
53          }
54
55      return 0;
56    }
```

B.2.2 Client program

```
1
2     #include <OB/CORBA.h>
3     #include <OB/Util.h>
4     #include <person_impl.h>
5
6
7     #include <fstream.h>
8
9     #ifdef TIME
10    #include "timer.h"
11    #endif
12
```

```
13  main(int argc, char * argv[])
14  {
15
16  try
17    {
18
19  #ifdef TIME
20        struct timeval time1, time2;
21        struct timezone tz;
22
23        cout <<"\nStarting Timing";
24        gettimeofday(&time1, &tz);
25        cout <<"\nStarting Time: " << time1.tv_sec <<"." << time1.tv_usec;
26  #endif
27
28    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
29    const char* refFile = "person.ref";
30    ifstream in(refFile);
31    char s[1000];
32    in >> s;
33    CORBA_Object_var obj = orb->string_to_object(s);
34    assert (!CORBA_is_nil(obj));
35
36    Person_var per = Person::_narrow(obj);
37    assert (!CORBA_is_nil(per));
38
39    CORBA_String_var buf;
40    int n,val=0;
41    for (n=0;n<10000;n++)
42      {
```

```
43          buf = CORBA_string_dup(per->GetStr());
44        }
45
46      cout <<"\nBuffer's size is: " << strlen(buf) << endl;
47
48    #ifdef TIME
49          gettimeofday(&time2, &tz);
50          cout <<"\nEnding Time: " << time2.tv_sec <<"." << time2.tv_usec;
51          char timediff[50] = {'\0'};
52          timer(time1, time2, timediff);
53
54          cout <<"\nTotal time taken for client call: " << timediff;
55    #endif
56
57      }
58    #ifdef __GNUG__
59        catch(CORBA_COMM_FAILURE& ex)
60    #else
61        catch(CORBA_SystemException& ex)
62    #endif
63        {
64            OBPrintException(ex);
65            return 1;
66        }
67        return 0;
68    }
```

Appendix C

DCOM examples

C.1 Implementing the Component

```
1
2
3    #include <iostream.h>
4    #include <objbase.h>
5
6    #include "Iface.h"
7    #include "Registry.h"
8
9
10
11   // Global Vars
12
13   static HMODULE g_hModule = NULL ;  // DLL Module Handle
14   static long g_cComponents = 0;     // Count of active components
15   static long g_cServerLocks = 0;          // Count of Locks
16
```

```
17  // Name of component
18  const char g_szFriendlyName[] = "Person" ;
19
20  // Version-independent ProgID
21  const char g_szVerIndProgID[] = "Inproc.SimpleCase" ;
22
23  // ProgID
24  const char g_szProgID[] = "Inproc.SimpleCase.1" ;
25
26  //Component
27
28  class CPerson : public IAge
29  {
30  public:
31    virtual HRESULT __stdcall QueryInterface (const IID & iid,
32                                              void ** ppv);
33    virtual ULONG __stdcall AddRef();
34    virtual ULONG __stdcall Release();
35
36    //Interface Iage
37    virtual void __stdcall SetAge(int num)
38    {
39      cout << "\nIn IAge interface, SetAge method.";
40      age = num;
41    }
42
43    virtual int __stdcall GetAge()
44    {
45      cout << "\nIn IAge interface, GetAge method.";
46      return age;
```

```
47      }
48      //Constructor
49      CPerson();
50
51      //Destructor
52      ~CPerson();
53
54    private:
55      long m_cRef; //Reference count
56      int age;      //Data member for age;
57    };
58
59    //Constructor body
60    CPerson::CPerson() : m_cRef(1)
61    {
62      age=0;
63      InterlockedIncrement(&g_cComponents) ;
64    }
65    CPerson::~CPerson()
66    {
67            InterlockedDecrement(&g_cComponents) ;
68            cout << "\nComponent:\t\tDestroy self." ;
69    }
70
71    //IUnknown implementation for our component
72
73    HRESULT __stdcall CPerson::QueryInterface(const IID & iid,
74                                              void ** ppv)
75    {
76
```

```
77      if ( (iid == IID_IUnknown) || (iid == IID_IAge) )
78        {
79          *ppv = static_cast<IAge*>(this);
80        }
81      else
82        {
83          *ppv = NULL;
84          return E_NOINTERFACE ;
85        }
86      reinterpret_cast<IUnknown*>(*ppv)->AddRef();
87      return S_OK ;
88    }
89
90    ULONG __stdcall CPerson::AddRef()
91    {
92      return InterlockedIncrement (&m_cRef) ;
93    }
94
95    ULONG __stdcall CPerson::Release()
96    {
97      if (InterlockedDecrement(&m_cRef) == 0)
98        {
99          delete this ;
100         return 0 ;
101       }
102     return m_cRef ;
103   }
104
105   //Class Factory
106
```

```
107  class CFactory : public IClassFactory
108  {
109  public:
110    virtual HRESULT __stdcall QueryInterface(const IID &iid,
111                                             void ** ppv);
112    virtual ULONG __stdcall AddRef();
113    virtual ULONG __stdcall Release();
114
115    //Interface IClassFactory
116    virtual HRESULT __stdcall CreateInstance(IUnknown * pUnknownOuter,
117                                             const IID & iid,
118                                             void ** ppv);
119    virtual HRESULT __stdcall LockServer (BOOL bLock);
120
121    //Constructor
122    CFactory() : m_cRef(1) {}
123    //Destructor
124    ~CFactory () { cout << "\nClass factory:\t\tDestroy self."; }
125
126  private:
127    long m_cRef;
128  };
129
130  //Class Factory IUnknown implementation
131
132  HRESULT __stdcall CFactory::QueryInterface(const IID& iid,
133                                             void ** ppv)
134  {
135    if ((iid == IID_IUnknown) || (iid == IID_IClassFactory) )
136      {
```

```
137         *ppv = static_cast<IClassFactory*>(this) ;
138       }
139     else
140       {
141         *ppv = NULL;
142         return E_NOINTERFACE ;
143       }
144
145     reinterpret_cast<IUnknown*>(*ppv)->AddRef();
146     return S_OK;
147   }
148
149   ULONG __stdcall CFactory::AddRef()
150   {
151           return InterlockedIncrement(&m_cRef) ;
152   }
153
154   ULONG __stdcall CFactory::Release()
155   {
156           if (InterlockedDecrement(&m_cRef) == 0)
157           {
158                   delete this ;
159                   return 0 ;
160           }
161           return m_cRef ;
162   }
163
164   // IClassFactory implementaton
165   HRESULT __stdcall CFactory::CreateInstance(IUnknown * pUnknownOuter,
166                                              const IID & iid,
```

```
167                                                 void ** ppv)
168 {
169   cout << "\nClass factory:\t\tCreate Component.";
170
171   // Cannot aggregate
172   if (pUnknownOuter != NULL)
173     return CLASS_E_NOAGGREGATION;
174
175   // Create component
176   CPerson * pPer = new CPerson;
177   if (pPer == NULL)
178     return E_OUTOFMEMORY;
179
180   HRESULT hr = pPer->QueryInterface(iid, ppv);
181
182   //Release IUnkown pointer.
183
184   pPer->Release();
185
186   return hr;
187 }
188
189 HRESULT __stdcall CFactory::LockServer(BOOL bLock)
190 {
191   if (bLock)
192     {
193       InterlockedIncrement(&g_cServerLocks) ;
194     }
195   else
196     {
```

```
197         InterlockedDecrement(&g_cServerLocks) ;
198       }
199     return S_OK ;
200   }
201
202   // Exported functions
203   //
204
205   //
206   // Can DLL unload now?
207   //
208   STDAPI DllCanUnloadNow()
209   {
210           if ((g_cComponents == 0) && (g_cServerLocks == 0))
211           {
212                   return S_OK ;
213           }
214           else
215           {
216                   return S_FALSE ;
217           }
218   }
219
220   // Get Class factory (Entry Point)
221
222   STDAPI DllGetClassObject (const CLSID & clsid,
223                             const IID & iid,
224                             void ** ppv)
225   {
226     cout << "\nDllGetClassObject:\tCreate class factory for person.";
```

```
227
228     //Can we create this component?
229     if (clsid != CLSID_PersonComponent)
230       {
231         return CLASS_E_CLASSNOTAVAILABLE;
232       }
233
234     CFactory* pFactory = new CFactory;
235     if (pFactory == NULL)
236       {
237         return E_OUTOFMEMORY;
238       }
239
240     //Get required interface.
241     HRESULT hr = pFactory->QueryInterface(iid, ppv);
242     pFactory->Release();
243
244     return hr;
245   }
246
247   //Server Registration
248   STDAPI DllRegisterServer()
249   {
250           return RegisterServer(g_hModule,
251                                 CLSID_PersonComponent,
252                                 g_szFriendlyName,
253                                 g_szVerIndProgID,
254                                 g_szProgID) ;
255   }
256
```

```
257
258  //
259  // Server unregistration
260  //
261  STDAPI DllUnregisterServer()
262  {
263          return UnregisterServer(CLSID_PersonComponent,
264                                  g_szVerIndProgID,
265                                  g_szProgID) ;
266  }
267
268  BOOL APIENTRY DllMain(HANDLE hModule,
269                        DWORD dwReason,
270                        void* lpReserved)
271  {
272          if (dwReason == DLL_PROCESS_ATTACH)
273          {
274                  g_hModule = hModule ;
275          }
276          return TRUE ;
277  }
```

C.2 Implementing the Client

```
1    #include <iostream.h>
2    #include <objbase.h>
3    #include "Iface.h"
4
5    int main()
6    {
```

```
7
8       //Initialize COM library
9       CoInitialize(NULL);
10      cout << "\nCall CoCreateInstance to create person.";
11      cout << "component\n and get IAge Interface.";
12      IAge * pIAge = NULL;
13      HRESULT hr = ::CoCreateInstance(CLSID_PersonComponent,
14                                      NULL,
15                                      CLSCTX_INPROC_SERVER,
16                                      IID_IAge,
17                                      (void**)&pIAge) ;
18      if (SUCCEEDED(hr))
19        {
20          cout <<"\nGot IAge interface from Person.";
21          /*
22          int num;
23
24          cout <<"\nEnter age to set: ";
25          cin >> num;
26          pIAge->SetAge(num);
27          */
28
29          cout << "\n\nAge set to: " << pIAge->GetAge();
30          cout << "\nReleasing IAge interface";
31          pIAge->Release();
32        }
33      else
34        {
35          cout << "\nClient: Could not create componet. hr = "
36               << hex << hr << endl;
```

```
37        }
38      //Uninitialize COM library
39      CoUninitialize();
40
41      return 0;
42    }
```

Biography of the Author

Santosh Sreenivasan was born to Mr. Narayanan (Chinu) and Lalita Sreenivasan in Ahmedabad, India on September 3, 1975. The first seven years of his life were spent in different parts of India, as his family moved to Assam, in the Northeast corner of the country, and later came back to Ahmedabad. In 1983, all three of them moved to the United States of America, as Santosh's father struck out on a bold career initiative. Santosh spent the years of childhood in East Brunswick, New Jersey, where he also graduated from high school in 1992. Deciding to return to India for his undergraduate studies, Santosh acquired his Bachelor's degree in Computer Science from Loyola College in Madras in 1996. These years in Madras forever changed Santosh's life, bringing him to the feet of his guide, Shri Parthasarathi Rajagopalachari. Santosh returned to the United States in 1996 to pursue a Master's degree in Computer Science at Lehigh University in Bethlehem, Pennsylvania. Having acquired this degree in January 1998, Santosh is setting out on the path ahead, wherever it may lead.

END

OF

TITLE